




IG-Tree: an efficient spatial keyword index for planning best path queries on road networks

Anasthasia Agnes Haryanto¹ · Md. Saiful Islam²  · David Taniar¹ · Muhammad Aamir Cheema¹

Received: 30 July 2018 / Revised: 6 September 2018 / Accepted: 31 October 2018 /
Published online: 15 November 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract

Due to the popularity of Spatial Databases, many search engine providers have started to expand their text searching capability to include geographical information. Because of this reason, many new queries on spatial objects affiliated with textual information, known as the Spatial Keyword Queries, have taken significant research interest in the past years. Unfortunately, most of existing works on Spatial Keyword Queries only focus on objects retrieval. There is barely any work on route planning queries, even though route planning is often needed in our daily life. In this research, we propose the Best Path Query, which we find the best optimum route from two different spatial locations that visits or avoids the objects that are specified by the textual data given by the user. We show that Best Path Query is an *NP-Hard* problem. We propose an efficient indexing technique, namely *IG-Tree*, and three different algorithms with different trade-offs to process the Best Path Queries on Road Networks. Our extensive experimental study demonstrates the efficiency and accuracy of our proposed approach.

Keywords Spatial databases · Spatial keywords · Trip planning queries · IG-Tree · Best path · Road networks

✉ Anasthasia Agnes Haryanto
agnes.haryanto@monash.edu

Md. Saiful Islam
mdsaiful.islam@griffith.edu.au

David Taniar
david.taniar@monash.edu

Muhammad Aamir Cheema
aamir.cheema@monash.edu

¹ Faculty of Information Technology, Monash University, Melbourne, Australia

² School of Information and Communication Technology, Griffith University, Gold Coast, Australia

1 Introduction

Spatial Databases have taken a big interest in today’s society. A lot of applications are using spatial data to help our daily necessity, such as the GPS. The application of spatial data itself is accessible not only on desktop computers, but it is also very popular in mobile environments these days as mobile devices provide significant services in our everyday life [1, 24, 29, 38, 39, 44, 50]. According to StatisticBrain, about 81% of mobile users use their device for Maps and directions [53]. With the popularity of Spatial Databases, search engine providers, such as Google and Yahoo, have broadened their text searching capability to provide geographical information [6, 18]. The GlobalWebIndex showed that Google Maps is the most used app as it is used by 54% of the general smartphone users [54]. In the past, conventional search engines could only process simple user queries, such as finding a Point of Interest (POI) based on a certain keyword [25]. However the high demand in spatial data compels the search engines to have the capability of processing more than such simple queries. A lot of new queries on spatial objects affiliated with textual information, known as Spatial Keyword Queries [45], have been studied in recent years. A number of researches have been done in order to improve the geographical search engines. Yet there are still many challenges to combine and process both the textual information and the spatial data.

Nowadays, each spatial object contains one or more meaningful keywords as to represent the object’s entities [15, 18]. The keywords may contain country name, city name, address, references to landmark, or even type of road [6]. For example in Figure 1, the points denote spatial objects and each object is affiliated with one or more keywords. Through the existence of this kind of spatial keywords information, the Spatial Keyword Queries become varied. Some of the commonly used queries are the *Top-k kNN Query*, *Boolean kNN Query*, and *Boolean Range Query* [7]. All of these queries require the user to give a spatial location

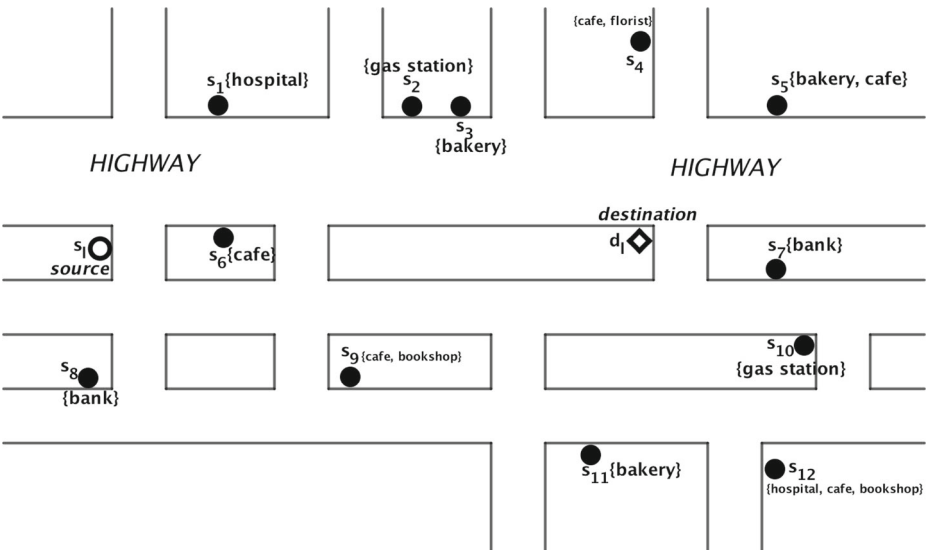


Figure 1 Illustration of a road network with spatial keyword objects

(normally the current user location) and textual data in the form of keywords as the input. While the output of these queries is spatial objects that are nearest to the user's location and contain the keywords given. The principle parameter used to identify the nearest objects is based on the shortest path distance, which is basically computing the minimum distance between two location points. Although the existing shortest path-based solutions are useful, they are not always sufficient for our needs. In real life, we often want to plan our trip with the most efficient cost (e.g. time, distance) taken. We may need to stop by several locations in our trip before arriving to the designated destination and there are also times when we would like to avoid some spatial objects that can interfere our activities. Planning a trip is eminently more complex than a simple source-to-destination type of query. Unfortunately, the existing studies on trip planning query in Spatio-Textual area are not flexible enough to answer this kind of query. Furthermore, often at times researchers only consider users to give keywords just to find POIs. But in reality, this is not always the case. Some query keywords may have negative connotations, such as traffic jams, which means that not all user given keywords can be considered as POI.

In this paper, we propose a new variant of *spatial keywords query*. Given a user with his/her location, this user wants to go to his/her destination while stopping by or avoiding several locations denoted by certain keywords. For instance a user wants to go to his workplace from his house, but before arriving to the workplace he wants to stop by a gas station to refill his car fuel, a bakery to get some breakfast, and also wish to avoid any highway along his trip (see Figure 1). In this query, the user specifies the source and destination locations and several other keywords. The keywords specified are *gas station*, *bakery*, and avoid *highway*. So we need to find the most optimum path for the user that satisfies his preferred keywords condition. Using the illustration in Figure 1, assume that s_l is the user's house (source location) and d_l is his workplace (destination location). There are a number of spatial objects that contain the keyword of *gas station* and *bakery*, such as s_2, s_3, s_5, s_{10} , and s_{11} , hence there are many possibilities of path combination to be established from s_l to d_l passing through at least one of each keyword. Looking at the road network, the path with the shortest distance is from s_l to s_2 to stop by a gas station, then s_3 to stop by a bakery, and then d_l . However this path passes through a highway. One of the keywords specified by the user to avoid is *highway*, which means that the path does not satisfy all the criteria given by the user. Therefore the best path that offers the least sum of distance and meets the criteria is from s_l to s_{11} for *bakery*, then s_{10} to stop by the *gas station*, then finally the destination d_l (obviously this path also avoids any *highway*). We call this kind of query as the *Best Path Query* (BP).

In Best Path Query, we are dealing with both spatial and textual data. Hence the user given query has two main parts: the spatial data part that consists of the source and destination locations that are given by the user, and the textual data part that consists of the keywords that the user would like to pass or avoid throughout his/her trip to the destination. Based on the user input, the query keyword itself can be classified into *positive* or *negative situations*. As in previous example, the user gave a negative keyword, which is to avoid highway. The positive keyword here is the POI that he wants to pass by, which are the gas station and bakery. The formal definition of Best Path Query is given as follow:

Definition 1 (Best Path Query) Given a source location s_l , a destination location d_l , and a set of keywords $K = \{k_1, k_2, \dots, k_n\}$, where each k_i for $1 \leq i \leq n$ can be positive (denoted by k^+) or negative (denoted by k^-), find the Best Path from s_l to d_l , denoted by $BP(s_l, d_l, K)$, that passes through all k^+ and avoid all k^- with optimum cost.

1.1 Challenges

The main challenge in this research is in the insufficiency of current solutions to trip planning in *spatial keywords area*, especially on Best Path Query. Existing studies do not take into consideration negative keywords. Having negative keywords actually increase the complexity of the problem as we have to make sure that we can avoid certain paths. There are definitely some cases where the result cannot be retrieved as we have to avoid all of the paths in between the source and destination locations.

The road networks are usually represented as a graph. The edges typically represent the road segments while the vertices represent the road intersections. So the path generation is always limited only to the edges to adjacent vertices. This increases the complexity of our query computation when we are working specifically in this environment. The complexity is also intensified with the types of query keywords given by the user. When the query keywords are negative, a lot of the paths will be blocked as we have to avoid them. We always have to make sure that we plan the route optimally despite these complexities.

Another challenge is that most of the route planning problems are regarded as a generalization of Traveling Salesman Problem (TSP) problem [3, 4]. They are *NP*-hard problems. The solutions offered for these queries are in polynomial time approximation algorithms. Even so, the solutions offered generally involve no pre-processing. This causes more computation particularly since the computation requires the processing of both spatial and textual relevancy. So having an on-the-go solution does not always guarantee performance efficiency. There are also very limited indexing techniques in the Spatio-Textual area, especially on road networks. In this research we attempt to provide a solution to this problem by offering a novel index that incorporate both *keywords* and *spatial road networks information* based on *G-Tree* [51, 52] and *IR²-Tree* [15].

1.2 Contributions

Our main contributions in this paper are as follows:

- We formally define Best Path problem on road networks and prove that this is an *NP*-hard problem.
- We develop a novel indexing scheme, called *IG-Tree*, for planning Best Path queries in road networks.
- We present three approximate algorithms with different trade-offs for searching Best Paths on road networks.
- We also demonstrate the effectiveness and efficiency of our algorithms through comprehensive experiments on real datasets.

1.3 Organisation

The rest of the paper is organised as follows. Section 2 presents the preliminaries and the query model for Best Path problem on road networks. Section 3 discusses the computational complexities of the Best Path problem on road networks. We introduce the *IG-Tree* in Section 4 and discuss our algorithms to solve the Best Path problem in Section 5. Section 6 presents the experimental evaluations of all the algorithms proposed. We discuss some related works in Section 7. Finally, Section 8 concludes the paper.

2 Preliminaries

This section presents the necessary background information, and the data and query model for *Best Path* problem on road networks.

2.1 Road network

We consider road network as an undirected weighted graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Each edge $(u, v) \in E$ connects two adjacent vertices $u, v \in V$ and is associated with a non-negative weight $w(u, v) > 0$ that represents distance or travel time.

A path $P(v_1, v_n) = \{v_1, v_2, \dots, v_n\}$ is a sequenced vertices such that v_i is adjacent to v_{i+1} , i.e., $(v_i, v_{i+1}) \in E$, for $1 \leq i < n$. The cost of a path P , denoted by $cost(P)$, is the sum of weights of the edges of P . Given vertices u and v , we use $\delta(u, v)$ to denote the shortest path from u to v while we use $dist(u, v)$ to denote the cost of $\delta(u, v)$. Figure 2 shows an example of a road network. If given a source vertex v_1 and destination vertex v_4 , then $\delta(v_1, v_4) = \{v_1, v_2, v_5, v_4\}$ is the shortest path between v_1 and v_4 and $dist(v_1, v_4) = 6$.

2.2 Data model

A spatio-textual object o is an object with a spatial location from $S = \{s_1, s_2, \dots, s_m\}$ that contains a set of keywords from $T = \{t_1, t_2, \dots, t_x\}$. We assume that spatio-textual objects are located at vertices in V . The weight of an edge $(u, v) \in E$ is the travel time or road network distance of two spatially-adjacent objects o_1 and o_2 representing u and v , respectively. We use vertex v or spatio-textual object o interchangeably in this paper.

2.3 Query model

Given a road network G , the user queries consist of a source location s_l , a destination location d_l , and preferred set of keywords $K = \{k_1, k_2, \dots, k_n\}$, where each $k_i, 1 \leq i \leq n$, can be positive (k^+) or negative (k^-). A positive keyword k^+ means that the keyword satisfies what the user wants, while a negative keyword k^- means that part of the keyword expresses negative connotation which the user wants to avoid. We assume $K \subseteq T$. Having all these information, we want to find the Best Path $BP(s_l, d_l, K)$ that establishes a shortest path

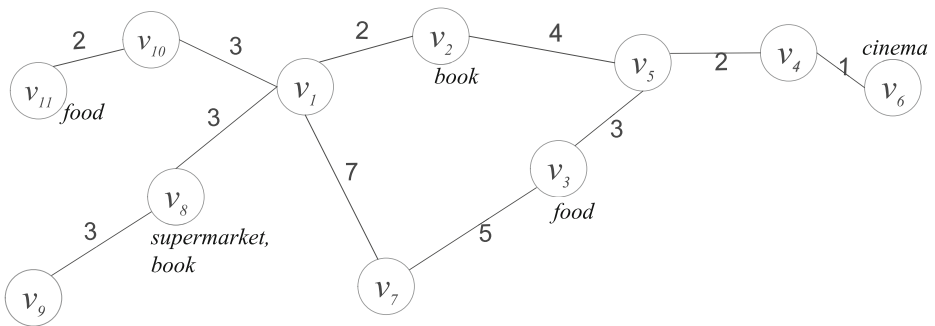


Figure 2 Example of a road network

from s_l to d_l , and that passes through all k^+ and avoids all k^- keyword matching vertices in G .

Table 1 presents the list of mathematical notations used in this paper.

3 Complexity analysis

The Best Path problem is different from the general Shortest Path problem. In Best Path problem, we want to find the path with objects of interest along our way to the destination. The objects of interest are determined by the keywords. The result of Best Path itself often is longer in distance than the Shortest Path, but there are also cases where it can have the same result as the Shortest Path. When the user does not specify any keywords at all in the query, then the Best Path is basically the Shortest Path since there are only source and destination locations provided. The Shortest Path problem is solvable in polynomial time. Therefore we can implement the commonly used algorithms for Shortest Path, such as Dijkstra algorithm, for this particular case.

Lemma 1 Given s_l, d_l , and $K = \{\}$, $BP(s_l, d_l, K) = \delta(s_l, d_l)$.

Proof Base Case: If $|P| = 1$ and $K = \{\}$, then $P = \{s_l\}$ and $cost(P) = 0 = dist(s_l, s_l)$. Hence, $\delta(s_l, s_l) = BP(s_l, s_l, K)$.

Inductive hypothesis: Let u be the last vertex added to P , $P' = P \cup \{u\}$. In this case our Inductive Hypothesis is

$$\text{for each } y \in P', \text{cost}(P'(s_l, y)) = \text{cost}(\delta(s_l, y))$$

Inductive step: Suppose that there is a shortest path Q from s_l to u and

$$\text{cost}(Q) < \text{cost}(P'(s_l, u))$$

Since Q is a shortest path, then $\text{cost}(Q) = dist(s_l, u)$.

Assume that the shortest path Q begins at P' and then leaves P' before arriving to the destination u . (y, z) is the first edge in Q that leaves P' , and Q_y is a shortest path from s_l to y , so

$$\text{cost}(Q_y) + w(y, z) \leq \text{cost}(Q)$$

Table 1 List of notations used in the paper

Notation	Definition
G	Road network
$\delta(u, v)$	Shortest path between u and v
$dist(u, v)$	Distance between u and v
s_l, d_l	Source location, destination location
K	Query keywords given by user
k^+, k^-	Positive keyword, negative keyword
$BP(s_l, d_l, K)$	The Best Path from s_l to d_l that passes through all k^+ and avoids k^-

Since according to the Inductive Hypothesis $cost(P'(s_l, y))$ is also the cost of $\delta(s_l, y)$, then $cost(P'(s_l, y)) \leq cost(Q_y)$. So it gives us

$$cost(P'(s_l, y)) + w(y, z) \leq cost(Q_z)$$

As y and z are adjacent vertices, then

$$cost(P'(s_l, z)) \leq cost(P'(s_l, y)) + w(y, z)$$

Since u is part of Q , so

$$cost(P'(s_l, u)) \leq cost(P'(s_l, z))$$

Therefore shortest path Q does not exist, so $cost(P'(s_l, u)) = \delta(s_l, u) = BP(s_l, u, K)$. \square

However, this is not the same when we have a keyword specified by the user. Depending on the positive or negative value, the path may or may not be retrieved. If the query contains only one positive keyword, doing the shortest path search from source to the nearest vertex that has matching keyword then doing another shortest path search from the nearest vertex with matching keyword to the destination is incorrect. As an example using the road network in Figure 1, assume that the source s_l is v_5 , destination d_l is v_{10} , and then the preferred keyword given by the user is located at v_1 and v_3 . If we choose the nearest keyword match vertex from v_5 , v_3 is the nearest since $dist(v_5, v_3)$ is 3, while $dist(v_5, v_1)$ is 6. However if we calculate the total distance from v_5 to v_3 to v_{10} , the total is 18. On the contrary, the total distance from v_5 to v_1 to v_{10} is 9, which is a lot shorter than having v_3 as the chosen keyword match vertex. Hence, choosing the nearest vertex with matching keyword will cause local minimum problem.

Meanwhile if the query keyword does not exist in T , then $BP(s_l, d_l, K)$ is also $\delta(s_l, d_l)$.

Lemma 2 Given s_l, d_l , and $K = \{k_1\}$, $k_1 \notin T$. Thus $BP(s_l, d_l, K) = \delta(s_l, d_l)$.

Proof If $k_1 \notin T$, then $K = \{\}$; which is already proven in Lemma 1. \square

If the user query contains a negative keyword, then the vertices that have this particular keyword need to be avoided/blocked. When we do the query processing to find the Best Path, these vertices can be pruned/disconnected from the graph as they are no longer considered as POI. This may also cause a deadend in the graph since a potential path can be ceased with the disappearance of a vertex. So when an edge of a vertex with negative keyword is a bridge, we will not be able to retrieve any Best Path. Lemma 3 and 4 prove the non-existence of Best Path for this particular case.

Lemma 3 If there exists a bridge (u, v) in G , and G consists of subgraph H and I that are connected by (u, v) . Given $s_l \in H$ and $d_l \in I$, then $(u, v) \subseteq BP(s_l, d_l, K)$.

Proof Assume that (u, v) is a bridge in G and $BP(s_l, d_l, K)$ on G does not contain (u, v) . Since $BP(s_l, d_l, K)$ is a path that every vertex in it has to be connected with each other, and $BP(s_l, d_l, K) \subseteq G \setminus \{(u, v)\}$ which $G \setminus \{(u, v)\}$ is a disconnected graph, then it must be disconnected. \square

Definition 2 A critical path $cp(v_i, v_j)$, $i \neq j$, is a path that consists of one or more graph bridges between v_i and v_j .

Lemma 4 Given $s_l, d_l, K = \{k^-\}$, and $BP(s_l, d_l, K) = cp(s_l, d_l)$. Then $BP(s_l, d_l, K)$ does not exist.

Proof Suppose that there exists a bridge (u, v) in $BP(s_l, d_l, K)$ and $k^- \in (u, v)$. Since we have to avoid negative keywords, then (u, v) has to be pruned from $BP(s_l, d_l, K)$. Hence, the graph is now disconnected as proved in Lemma 3. \square

Even though negative keywords can cause path blockage, it does not mean that we cannot retrieve any Best Path at all. The path blockage might cause us to re-route to another path even though it may cause a longer path.

Lemma 5 Given d_l, s_l , and $K = \{k^-\}$, if $BP(s_l, d_l, K) \neq cp(s_l, d_l)$, then there exists $BP(s_l, d_l, K)$.

Proof We can establish a path since the graph is still connected even though there is k^- . \square

In the case where the user gave a set of positive keywords as the query input and there is no negative keyword at all, then the Best Path's result will be similar to the state-of-the-art Trip Planning Route Queries (TPQ)'s result [27].

Lemma 6 Given d_l, s_l , and $K = \{k_1^+, k_2^+, \dots, k_n^+\}$, $BP(s_l, d_l, K) = TPQ$.

Proof When all keywords are positive, then by definition, Best Path is the same as TPQ where we have to find the best trip/route from s_l , passing through one point from each category and then ending the trip at d_l . \square

Looking at Lemma 6, it means that Best Path also can be considered as NP -hard problem. Thus in the following Lemma we try to reduce Best Path problem to Traveling Salesman Problem (TSP), to which TSP is a well-known NP -hard problem.

Lemma 7 $BP(s_l, d_l, K)$ is NP -hard.

Proof Assume a road network G of a set of spatio-textual objects with spatial locations $S = \{s_1, s_2, \dots, s_m\}$ and each object has a distinct keyword from the keyword set T . Moreover, the user queries consist of a source location s_l , destination location d_l , and preferred keywords $K = \{k_1, k_2, \dots, k_n\}$. Again, assume that for K , we need to visit every vertex in G . Now, we reduce the Best Path Problem to TSP. Let $G^{prime} = (V^{prime}, E^{prime})$ as the instance of TSP, where $V' = V$ and $E' = (u, v)$ for any $u, v \in V'$. Then for road network G , we complete the graph by connecting all vertices. The cost function between G and G' is as follow:

$$cost(u, v) = \begin{cases} 0, & \text{if } edge(u, v) \in E \\ 1, & \text{if } edge(u, v) \notin E \end{cases}$$

Suppose that Best Path $BP(s_l, d_l, K)$ exists in G and has cost ≤ 0 in G' , hence there exists a solution to TSP in G' with cost ≤ 0 . \square

4 Data index

This section presents the *IG-Tree*, an indexing technique for planning Best Path Queries on Road Networks. Before presenting the *IG-Tree*, we first provide a brief background on *G-Tree* [51, 52] and *IR²-Tree* [15], which are the indexing techniques that inspired us to develop the *IG-Tree* for processing $BP(s_l, d_l, K)$ queries efficiently.

4.1 G-Tree

One of the most efficient indexing techniques on Road Networks is the *G-Tree* [51, 52]. In *G-Tree*, the road network is partitioned recursively into sub-networks. The nodes in *G-Tree* correspond to a single sub-network and each node contains two or more road network vertices. The graph partition process is performed by using the multi-level partitioning algorithm [26], which guarantees that each subgraph will be of almost the same size. Figure 3 shows an example of graph partitioning of the road network given Figure 2. Here, the original graph is partitioned into two subgraphs, which are shown by G_1 and G_2 . In the next level, G_1 is partitioned again into two equal-sized subgraphs G_3 and G_4 . Similarly, G_2 is partitioned again into subgraphs G_5 and G_6 .

The vertices that connect two sub-networks together are marked as *borders* and they are stored in the *G-Tree* nodes. Figure 3 shows the example where vertex v_1 is the border of G_3 since it connects partition G_3 with other partitions G_4 and G_5 . The border in partition G_4 consists of v_8 and v_7 , the border in partition G_5 consists of v_2 and v_3 , and the border in partition G_6 is v_4 . In the partition G_1 , the borders are v_1 and v_7 as both connects G_1 with G_2 . While the borders in G_2 are v_2 and v_3 .

G-Tree does not store the distance of every vertex but stores the set of borders and the shortest path distance between borders that is kept in the distance matrix. For example for partition G_3 , the border is v_1 , so the distance matrix will contain the shortest path from v_1 to all other vertices in the subgraph partition $\{v_1, v_{10}, v_{11}\}$. The distance matrix itself is proven to be very efficient in terms of processing the k NN search on road networks [51, 52].

Though *G-Tree* is very efficient in indexing and processing nearest neighbor (NN), k nearest neighbor (k NN) and keyword-based k NN queries on road networks, it is not applicable for processing Best Path queries.

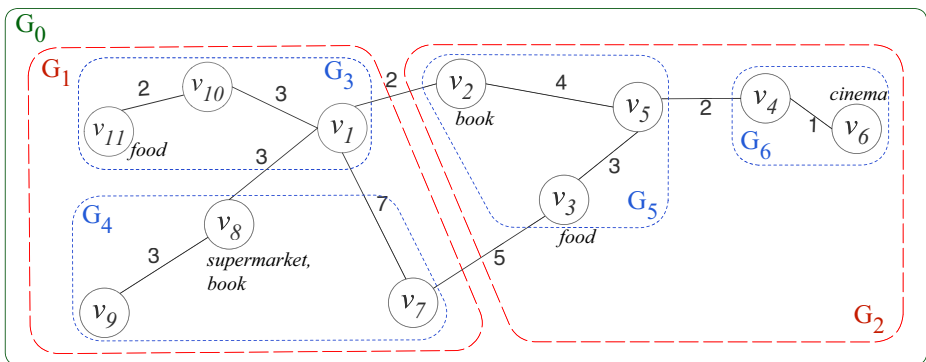


Figure 3 Graph partitioning on road network given in Figure 2

4.2 IR^2 -Tree

There are a number of indexing techniques proposed for processing *Spatial Keywords Queries*, one of them is the IR^2 -Tree. The IR^2 -Tree is first introduced by Felipe et al. [15]. It is a hybrid indexing approach that combines the R -Tree [5] and information retrieval signature files. However, this indexing technique is only applicable for spatial data objects in Euclidean space.

The indexing in IR^2 -Tree is performed by attaching the inverted index to the R -Tree, i.e., every tree node in IR^2 -Tree holds the information for both spatial location and keywords. The leaf nodes contain the actual spatial data and keywords. For example, assume that an object o_1 that contains keyword *book* is located in leaf node N_1 with spatial location of [38, 4] [93, 9] (upper right and bottom left coordinate of the *minimum bounding rectangle* (MBR)), while an object o_2 with keyword *supermarket* is located in leaf node N_2 with spatial location of [8, 15] [41, 32]. Suppose that the inverted index for keyword *book* is 10 and the inverted index for keyword *supermarket* is 01. Thus the leaf node N_1 contains the information of spatial location [38, 4] [93, 9] and keyword index 10, while leaf node N_2 contains the information of spatial location [8, 15] [41, 32] and keyword index 01.

As the leaf node in IR^2 -Tree stores the spatial data and keyword index, the non-leaf node contains the combination of several objects. The spatial information is based on the MBR, while the inverted index of the keywords are calculated using logical OR [15]. For example, the leaf nodes N_1 and N_2 from the previous example have the same parent node N_0 . So in this case, N_0 contains keyword information of 11 as this node consists of both keywords from N_1 and N_2 .

4.3 Proposed data index: IG-Tree

As previously discussed, the IR^2 -Tree [15] is used for indexing Spatial Keywords Queries in Euclidean space, while the G -Tree [51, 52] is used for indexing Road Networks. As Best Path is a type of Spatial Keywords Query on Road Network, each one of these indexing techniques has its own benefit to Best Path Query. Thus, we adopt these two indexing techniques to develop a new indexing scheme that can improve the processing of Best Path query: IG -Tree, a hybrid between IR^2 -Tree and G -Tree.

Using the road network in Figure 2, we attempt to create the IG -Tree. So following the graph partition technique used in G -Tree, we partition the graph into smaller subgraphs. Figure 3 shows the graph partitioning of the example road network given in Figure 2. The graph is divided into equal-sized subgraphs using the multi-level partitioning algorithm [26] and each partition consists of two or more vertices. At the leaf level of the tree, the subgraph G_3 consists of vertices v_1 , v_{10} and v_{11} ; subgraph G_4 consists of vertices v_7 , v_8 and v_9 ; subgraph G_5 consists of vertices v_2 , v_3 and v_5 ; and finally, subgraph G_6 consists of vertices v_4 and v_6 . Each partition makes up one node in the IG -Tree, as presented in Figure 4.

After the graph partition, we mark the borders of each partition. Borders are the vertices in one partition that are connecting the road network to another partition. For example the border for partition G_3 is v_1 since v_1 connects the subgraph to partition G_4 and partition G_5 . So the borders of G_4 are v_7 and v_8 ; the borders of G_5 are v_2 , v_3 and v_5 ; while the border of G_6 is v_4 . Based on these borders, we create the distance matrices. So the shortest path distances for every border in every node are pre-computed and stored in the matrices. Tables 2, 3, 4, 5, 6, 7, and 8 show the distance matrices for each node in IG -Tree.

Another aspect of the graph in Figure 3 is the keywords. Some vertices contain one or more keywords, thus we also need to index these keywords. The keywords can be turned

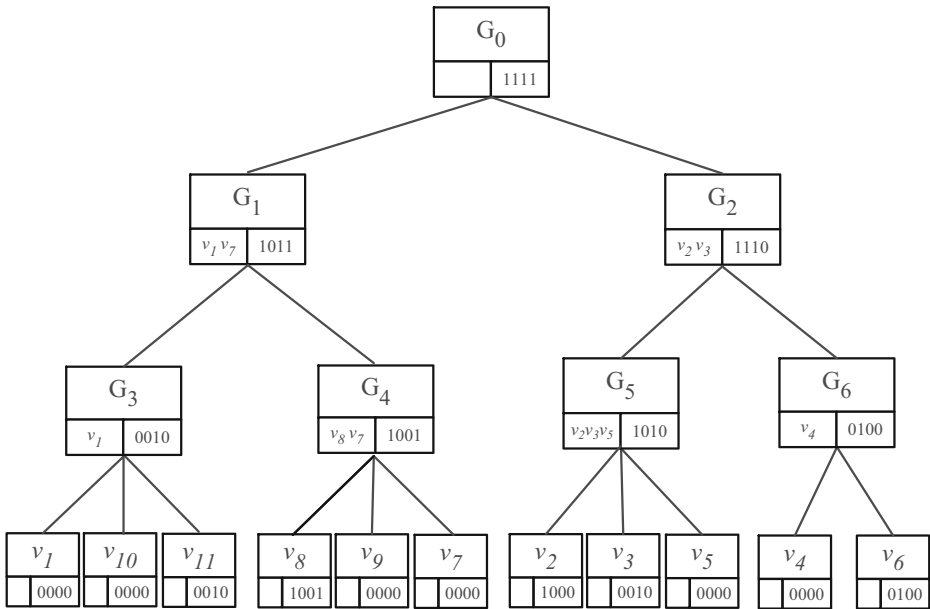


Figure 4 IG-Tree

Table 2 Distance Matrix for G_0

G_0	v_1	v_7	v_2	v_3
v_1	0	7	2	9
v_7	7	0	9	5
v_2	2	9	0	7
v_3	9	5	7	0

Table 3 Distance Matrix for G_1

G_1	v_1	v_8	v_7
v_1	0	3	7
v_8	3	0	10
v_7	7	10	0

Table 4 Distance Matrix for G_2

G_2	v_2	v_3	v_5	v_4
v_2	0	7	4	6
v_3	7	0	3	5
v_5	4	3	0	2
v_4	6	5	2	0

Table 5 Distance Matrix for G_3

G_3	v_1	v_{10}	v_{11}
v_1	0	3	5

Table 6 Distance Matrix for G_4

G_4	v_8	v_9	v_7
v_8	0	3	10
v_9	3	0	13
v_7	10	13	0

into inverted list. So the first step is to sort all of the keywords in the graph. Then for each keyword, we assign a binary value based on its existence in each node. For instance vertex v_2 contains only keyword *book*, thus the inverted list for v_2 is 1000. For node v_8 , it contains both keyword *book* and *supermarket*, so its inverted list is 1001. The inverted index for all the vertices of the graph in Figure 3 is presented in Table 9.

Based on the above inverted index list, we attach each inverted index to its corresponding vertices at the leaf nodes. For each parent node, its inverted index is calculated using logical OR of its child nodes. For instance G_3^{prime} 's inverted index is the result of logical OR of the inverted index of v_1, v_{10}, v_{11} . The result of 0000 or 0000 or 0010 is 0010, thus G_3^{prime} 's inverted index is 0010. The same calculation is applied for every non-leaf nodes. The root node will normally have all 1s for the index.

Even though we have indexed all of the available keywords and assign them to each node in the tree, having these indexes are not adequate. The inverted index only identifies that a certain keyword exists on a node but do not exactly identify the location until we go to the leaf node. Therefore we propose a Keyword Distance Matrix for each node. This *Keyword Distance Matrix* contains the distance of the nearest keyword matching vertex from each border. By having this matrix, the keyword search computation is sped up as we do not need to compute the keyword distance in processing time. The Keyword Distance Matrices for the *IG-Tree* in Figure 4 are shown at Tables 10, 11, 12, 13, 14, 15 and 16.

Based on the above discussion, there are several important components to build an *IG-Tree*. For every non-leaf node in *IG-Tree* contains the partition name, the border of each partition, and the inverted index (using the logical OR of its child node). Each non-leaf node also contains two types of matrices, which are the Distance Matrix and Keyword Distance Matrix. For every leaf node, it contains the road network's vertex and inverted list of the corresponding vertex. We also keep the geographic coordinate location of each vertex in the leaf node.

4.3.1 Space complexity of the *IG-Tree*

Height The height of *IG-Tree* is similar to *G-Tree* [51, 52] which is $\mathcal{H} = \log_f \frac{|V|}{\tau} + 1$, where f is the number of partition for each graph/subgraph, $|V|$ is the number of vertices in the given (road network) graph G , and τ is the number of maximum vertices on leaf node's subgraph.

Table 7 Distance Matrix for G_5

G_5	v_2	v_3	v_5
v_2	0	7	4
v_3	7	0	3
v_5	4	3	0

Table 8 Distance Matrix for G_6

G_6	v_4	v_6
v_4	0	1

Table 9 Keyword index

<i>Keyword</i>	<i>Book</i>	<i>Cinema</i>	<i>Food</i>	<i>Supermarket</i>
v_1	0	0	0	0
v_2	1	0	0	0
v_3	0	0	1	0
v_4	0	0	0	0
v_5	0	0	0	0
v_6	0	1	0	0
v_7	0	0	0	0
v_8	1	0	0	1
v_9	0	0	0	0
v_{10}	0	0	0	0
v_{11}	0	0	1	0

Table 10 Keyword Distance Matrix for G_0

G_0	1000	0100	0010	0001
v_1	2	9	5	3
v_7	9	11	5	10
v_2	0	7	7	5
v_3	7	6	0	11

Table 11 Keyword Distance Matrix for G_1

G_1	1000	0100	0010	0001
v_1	2	∅	5	3
v_8	0	∅	8	0
v_7	10	∅	12	10

Table 12 Keyword Distance Matrix for G_2

G_2	1000	0100	0010	0001
v_2	0	7	7	∅
v_3	7	6	0	∅
v_5	4	3	3	∅
v_4	6	1	5	∅

Table 13 Keyword Distance Matrix for G_3

G_3	1000	0100	0010	0001
v_1	∅	∅	5	∅

Table 14 Keyword Distance Matrix for G_4

G_4	1000	0100	0010	0001
v_8	0	\emptyset	\emptyset	0
v_9	3	\emptyset	\emptyset	3
v_7	10	\emptyset	\emptyset	10

Number of nodes Like G -Tree, IG -Tree has only one node in level 0, which is the root. In an arbitrary level i of the tree, there are f^i internal nodes as the number of partition for each graph (at level 0)/subgraph (at level > 0) is s . As τ is the maximum number of vertices on leaf node’s subgraph, there are $\frac{|V|}{\tau}$ leaf nodes. As a result, the number of nodes in IG -Tree is $\mathcal{O}\left(\frac{f}{f-1} \cdot \frac{|V|}{\tau}\right) = \mathcal{O}\left(\frac{|V|}{\tau}\right)$ which is again similar to that of G -Tree.

Number of inverted lists A node in IG -Tree contains an inverted list representing the keywords covered in that node. As the number of nodes in an IG -Tree is $\mathcal{O}\left(\frac{|V|}{\tau}\right)$, the number of inverted lists is $\mathcal{O}\left(\frac{|V|}{\tau} + |V|\right)$. Thus, the space complexity of maintaining inverted lists in IG -Tree becomes $\mathcal{O}\left(\frac{|V|}{\tau} \cdot |T| + |V| \cdot |T|\right)$, where $|T|$ is the number of keywords covered in the whole road network G and $|V| \cdot |T|$ is the space complexity of the inverted lists for all vertices in V .

Number of borders If we assume the road network to be modeled as a planar graph, the number of borders on average in a node of level i is $\mathcal{O}\left(\log_2 f \cdot \sqrt{\frac{|V|}{f^{i+1}}}\right)$ as per the calculation conducted in [51, 52]. As there are f^i nodes in a level i , the number of border nodes in an arbitrary level i is $\mathcal{O}\left(\log_2 f \cdot \sqrt{\frac{|V|}{f^{i-1}}}\right)$. If we sum this measure from level 1 to height of the tree $\log_f \frac{|V|}{\tau} + 1$, the total number of borders in an IG -Tree is $\mathcal{O}\left(\frac{\log_2 f}{\sqrt{\tau}} |V|\right)$, which is again similar to the G -Tree under the planar graph assumption.

Distance matrices The total distance matrix size of all leaf nodes is $\mathcal{O}(\sqrt{\tau}|V| \cdot \log_2 f)$ and the total distance-matrix of non-leaf nodes is $\mathcal{O}\left(|V| \cdot \log_2^2 f \cdot \log_f \frac{|V|}{\tau}\right)$ as per the calculation conducted in [51, 52].

Keyword distance matrices The average number of borders in a leaf node of IG -Tree is $\mathcal{O}(\log_2 f \cdot \sqrt{\tau})$ [51, 52] and the total number of keywords in G is $|T|$. Thus the keyword distance matrix size in a leaf node is $\mathcal{O}(\log_2 f \cdot \sqrt{\tau} \cdot |T|)$. The total keyword distance matrix size of all leaf nodes becomes $\mathcal{O}\left(\frac{|V|}{\tau} \cdot \log_2 f \cdot \sqrt{\tau} \cdot |T|\right)$. Each internal node on level i generates $\mathcal{O}\left(\log_2 f \cdot \sqrt{\frac{|V|}{f^{i+1}}}\right)$ borders on average [51, 52]. Therefore,

Table 15 Keyword Distance Matrix for G_5

G_5	1000	0100	0010	0001
v_2	0	\emptyset	7	\emptyset
v_3	7	\emptyset	0	\emptyset
v_5	4	\emptyset	3	\emptyset

Table 16 Keyword Distance Matrix for G_6

G_6	1000	0100	0010	0001
v_4	\emptyset	1	\emptyset	\emptyset

the keyword distance matrix size of each node at level i is $\mathcal{O}\left(\log_2 f \cdot \sqrt{\frac{|V|}{f^{i+1}}} \cdot |T|\right)$. In *IG-Tree*, there are f^i nodes at level i , therefore keyword distance matrix size at level i is $\mathcal{O}\left(f^i \cdot \log_2 f \cdot \sqrt{\frac{|V|}{f^{i+1}}} \cdot |T|\right) = \mathcal{O}\left(\log_2 f \cdot \sqrt{|V|} f^{i-1} \cdot |T|\right)$. Thus the total keyword distance matrix size of non leaf nodes is $\mathcal{O}\left(\sum_{0 \leq i < \mathcal{H}} \log_2 f \cdot \sqrt{|V|} f^{i-1} \cdot |T|\right)$.

4.3.2 Index reconstruction for tree node with negative query keywords

In the Best Path Query, the user is allowed to give keywords as input of the query and the query keywords can be positive and negative. The positive keywords denote the spatio-textual objects that the user wants to visit, while the negative keywords denote the spatio-textual objects that the user wants to avoid along his/her trip. Even though *IG-Tree* contains textual information of spatial objects, we still have to check the textual relevancy between the spatial object with the query keywords given by the user in order to consider an object to be visited or avoided. For the objects that contain positive keywords, the *IG-Tree* can compute the path well with the help of the Distance Matrices. For example if we want to compute the path from v_5 to the nearest *book*, we can directly refer to the Keyword Distance Matrix in Table 15 to save up some time. But this is different when negative keywords exist. Even though *IG-Tree* is designed to improve path computation on finding the Best Path, it still has a weakness when there is a negative keyword found in the query given by the user. As previously mentioned in Section 3, a vertex that holds one or more negative query keywords must be pruned from the road network graph. This is due to the fact that this particular vertex holds a query that the user wants to avoid/block. Currently *IG-Tree* consists of Distance Matrices that store the shortest paths between borders. So when a vertex is pruned from the graph, the shortest path may also change. The existing index has to be modified considering a vertex is gone and the Distance Matrices are no longer storing accurate distances. The new Distance Matrices will replace the existing matrices during the query processing time of the query with the corresponding negative keywords. The modification however depends on the location of the vertex in the *IG-Tree*:

- **Case-C1.** If vertex v that contains k^- is the border and no other border exists for a node that we must visit, then no path can be established. In this case, v is considered as a bridge. This situation is already proved through Lemmas 3 and 4 in Section 3.
- **Case-C2.** If vertex v that contains k^- is the border and there is/are other border(s), then path reconstruction is needed. The path reconstruction will involve the whole tree node where v is located and also the borders on other nodes that are adjacent to v . For example if v_3 in Figure 3 contains k^- , then the path reconstruction occurs on the whole G_5 tree node and its adjacent borders. The adjacent borders in this case can be identified through parent nodes of G_5 , whether the parent nodes has v_3 as one of their borders. G_2 and G_0 are indeed sharing v_3 as their border, so the path reconstruction will involve these two tree nodes as well. As v_3 is pruned from the graph, v_3 is then omitted from the Distance Matrices of G_5 , G_2 and G_0 . The border-to-border distances of these matrices are also affected because of the omission of v_3 , thus the entire matrices has to be recalculated because of the changes in the shortest path between these borders. The

Table 17 Reconstructed Distance Matrix for G_5

G_5	v_2	v_5
v_2	0	4
v_5	4	0

path reconstruction itself can be obtained using Dijkstra algorithm. Tables 17, 18 and 19 shows the Distance Matrices after path reconstruction.

- **Case-C3.** If vertex v that contains k^- is in the leaf node (not the border), then distance has to be recalculated. The index recalculation for this case does not affect the whole tree, but only on the tree node where the vertex with negative keyword lies. Similar to the previous case, the path reconstruction can be obtained using Dijkstra algorithm. When v is in the leaf node, we can focus on its own subgraph partition as it does not affect the other partitions like in the previous case. For instance, assume that v_{10} in Figure 3 contains k^- . v_{10} is a leaf node as it does not connect any sub-networks. v_{10} is located in partition G_3 , thus only this partition will need to be reconstructed which is shown in Table 20.

5 Query processing

We propose three Best Path query processing algorithms that can be applied on *IG-Tree*, namely the Optimal Distance Approximation Search, Ancestry Priority Search, and the Euclidean-based Approximation. A baseline algorithm is also provided in this section. The baseline algorithm offers precise solution, while the other three proposed algorithms offer approximation solution with different trade-offs. In each subsection, we discuss on how each algorithm works and their trade-offs.

5.1 Baseline algorithm

In this section, we discuss on the baseline algorithm that can be used on *IG-Tree* to find the Best Path. This algorithm is able to compute the result of Best Path query accurately. The key/main idea is to find the permutation of all possible combinations of positive keywords and then compare them in order to find the one that has the most efficient cost (least distance).

As an example, assume that we want to find the best path from v_1 to v_4 while passing through *cinema* and *book*. In this case, $sl = v_1$, $dl = v_4$, and $keywords = \{cinema, book\}$. The first step here is to turn the preferred keywords into inverted index so

Table 18 Reconstructed Distance Matrix for G_2

G_2	v_2	v_5	v_4
v_2	0	4	6
v_5	4	0	2
v_4	6	2	0

Table 19 Reconstructed Distance Matrix for G_0

G_0	v_1	v_7	v_2
v_1	0	7	2
v_7	7	0	9
v_2	2	9	0

that we can check its relevancy with the inverted index in *IG-Tree*. The preferred keywords are *cinema* and *book*, therefore the inverted list is 1100 ($K = 1100$). Then we have to find the partition that contains the source and destination in the *IG-Tree*, where v_1 is located under the partition G_3 and v_4 is located under the partition G_6 . After the source and destination locations are found, then we can start finding the best path $BP(v_1, v_4, 1100)$ that visits the chosen keywords.

Scanning through every single vertex in the leaf node that holds inverted index of 1100. The inverted index of 1000 can be found at v_2 and v_8 , while the inverted index of 0100 can be found at v_6 . Knowing the exact locations of the keywords, we can do cartesian product between each set of keywords. In this case, the cartesian product will be between $\{v_2, v_6\}$ and $\{v_8, v_6\}$. Then based on the cartesian product, we have to get the permutation to help computing the path with the least distance. The permutations for this case consist of $\{v_2, v_6\}$, $\{v_6, v_2\}$, $\{v_8, v_6\}$, and $\{v_6, v_8\}$. Based on these permutations, we can find the shortest path from s_i to each permutation, and then from the permutation to d_l . In this case, we will have four possible paths: $v_1 \rightarrow v_2 \rightarrow v_6 \rightarrow v_4 = 10$, $v_1 \rightarrow v_6 \rightarrow v_2 \rightarrow v_4 = 22$, $v_1 \rightarrow v_8 \rightarrow v_6 \rightarrow v_4 = 16$, $v_1 \rightarrow v_6 \rightarrow v_8 \rightarrow v_4 = 32$. Algorithm 1 shows the pseudocode for shortest path search in *IG-Tree*. While calculating the shortest paths, we also need to keep track of the path with the least sum of distance. At the end, we will obtain the Best Path with the most accurate solution. For this example, the Best Path $BP(v_1, v_4, 1100) = v_1 \rightarrow v_2 \rightarrow v_6 \rightarrow v_4$ with the least total distance of 10.

This algorithm guarantees the accuracy of finding the Best Path on road networks. However since the Best Path query is an *NP-Hard* problem, this algorithm definitely runs in non-polynomial time especially on a large datasets. In our experiment, it can spend up to 17 hours merely to find the Best Path with 5 query keywords even in a very small datasets with only 100 vertices. This is certainly impossible to be applied for our daily use. The pseudo-code of the baseline algorithm is given in Algorithm 2.

5.2 Optimal distance approximation search

Because of the non-polynomial time complexity of the baseline algorithm, we propose an approximation algorithm to compromise the runtime. This algorithm is a lot faster than the baseline one but its result is not 100% accurate.

When multiple keywords are involved in the query, the complexity rises as we have to know all possible combinations of the keywords in order to get the most optimal solution

Table 20 Reconstructed Distance Matrix for G_3

G_3	v_1	v_{11}
v_1	0	\emptyset

(distance-wise). However when there is only one keyword involved in the query, the query can be retrieved in polynomial time. Thus in this approximation algorithm we utilize this situation in order to retrieve the multiple keywords query. The way this algorithm works is that for each query keyword given by the user, we find the best path between the source location to the query keyword and then to the destination. By getting the best path for each keyword, we can locate the best possible location of each keyword that will give the shortest distance of source-keyword-destination. Then after we have the best candidate of each keyword, we find the path from source location to its nearest candidate, then from the nearest candidate to its next nearest candidate. We keep doing this until all keywords are covered, then finishing the path to the destination location.

Algorithm 1 Shortest Path Search $\delta(s_l, d_l)$ on IG-Tree

Input: $IG - Tree, s_l, d_l, inverted\ file\ K_{if}$ in

Output: $\delta(s_l, d_l)$ out

```

1: Find partition  $G_{s_l}$  that contains  $s_l$  on  $IG - Tree$ 
2: Find partition  $G_{d_l}$  that contains  $d_l$  on  $IG - Tree$ 
3: Current source node  $v_{s_l} = s_l$ 
4: Current source node  $v_{d_l} = d_l$ 
5: while  $G_{s_l} \neq G_{d_l}$  do
6:   Find nearest border  $b_{s_l}$  from  $v_{s_l}$ 
7:   if  $b_{s_l}$  contains  $k^-$  then
8:     if There is other border then
9:       Path reconstruction for partition  $G_{s_l}$ 
10:    else
11:      return -1
12:    end if
13:  end if
14:   $dist(d_l, b_{s_l}) += dist(v_{s_l}, b_{s_l})$ 
15:   $v_{s_l} = b_{s_l}$ 
16:  Find nearest border  $b_{d_l}$  from  $v_{d_l}$ 
17:  if  $b_{d_l}$  contains  $k^-$  then
18:    if There is other border then
19:      Path reconstruction for partition  $G_{d_l}$ 
20:    else
21:      return -1
22:    end if
23:  end if
24:   $dist(d_l, b_{d_l}) += dist(v_{d_l}, b_{d_l})$ 
25:   $v_{d_l} = b_{d_l}$ 
26:   $G_{s_l} = \text{parent node of } v_{s_l}$ 
27:   $G_{d_l} = \text{parent node of } v_{d_l}$ 
28: end while
29: if  $G_{s_l} = G_{d_l}$  then
30:   Check Distance Matrix to get the shortest path  $dist(v_{s_l}, v_{d_l})$  from  $v_{s_l}$  to  $v_{d_l}$ 
31:    $dist(s_l, d_l) = dist(s_l, b_{s_l}) + dist(d_l, b_{d_l}) + dist(v_{s_l}, v_{d_l})$ 
32:   return  $\delta(s_l, d_l)$ 
33: end if

```

For example a user wants to find the best path from v_{10} to v_7 while passing through a bookstore and a cinema. In this case, $s_l = v_{10}$, $d_l = v_7$, and $K = \{book, cinema\}$. In order to find the best path, we have to transform the query keywords given by the user into inverted list. Since the keywords are $\{book, cinema\}$, thus the inverted list is $K_{if} = 1100$.

In this algorithm, we have to firstly find the locations of s_l and d_l . Looking at the *IG-Tree*, s_l is located within partition G_3 , while d_l is located within partition G_4 as shown in Figure 5. Now for each keyword k_n in K_{if} , we have to find the best path from $s_l - k_n - d_l$. Assume that the first keyword that we want to find its best path is *book* to which its inverted index is 1000. In the road network, there are actually several vertices that contain the keyword *book*. So we have to calculate the total shortest path distance for each keyword location and then finding the one that has the least amount of distance. A naive solution here is to find the shortest path between s_l to k_n and then add up the shortest path between d_l to k_n . Since our current keyword index is 1000, v_8 and v_2 have the same index. Hence we have to establish the shortest path $\delta(v_{10}, v_8) + \delta(v_8, v_7)$ and also $\delta(v_{10}, v_2) + \delta(v_8, v_2)$. The way the shortest path works is similar to the previous section. The best path distance for visiting v_8 is $dist(v_{10}, v_8) + dist(v_8, v_7) = 6 + 10 = 16$, while the best path distance for visiting v_2 is $dist(v_{10}, v_2) + dist(v_2, v_7) = 5 + 9 = 14$. Based on these calculations, v_2 has the best path from v_{10} to v_7 as depicted in Figures 6 and 7. Thus, we can store v_2 to a candidate queue Q_k as the candidate vertex to find the multiple keywords best path query. The same process also goes for keyword *cinema*. The inverted index of keyword *cinema* is 0100. There is only one vertex in the road network that contains 0100, which is v_6 . Therefore the best path is going to be based on $\delta(v_{10}, v_6) + \delta(v_8, v_6)$. Hence, v_6 can be stored to a candidate queue Q_k as another candidate vertex for finding the multiple keywords best path query, specifically for keyword *cinema*.

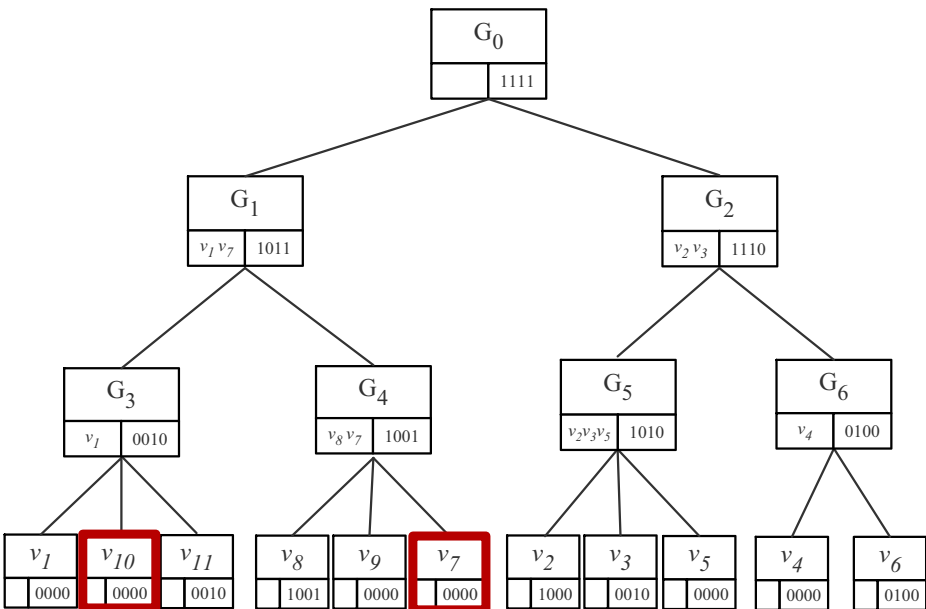


Figure 5 Finding location of v_{10} and v_7

Algorithm 2 The Baseline

Input: $IG - Tree, s_l, d_l, K$ in

Output: $BP(s_l, d_l, K)$ out

- 1: Index K to inverted file K_{if}
- 2: Find s_l on $IG - Tree$
- 3: Find d_l on $IG - Tree$
- 4: Scan through the leaf nodes to find vertices V_k with keywords that match to k^+ in K_{if} .
- 5: Find Cartesian Product $cart_product(V_k)$
- 6: **for** each $cart_product(V_k)$ result cp **do**
- 7: Find possible permutations of cp $Permutation(cp)$
- 8: **for** each permutation $perm$ of $Permutation(cp)$ **do**
- 9: $v_{start} \leftarrow 0$
- 10: **for** each vertex v in $perm$ **do**
- 11: **if** $v_{start} = 0$ **then**
- 12: Find $\delta(s_l, v)$;
- 13: **else**
- 14: Find $\delta(v_{start}, v)$;
- 15: **end if**
- 16: $v_{start} = v$
- 17: **end for**
- 18: Find $\delta(v_{start}, d_l)$;
- 19: **if** current path is the shortest **then**
- 20: Best Path $BP(s_l, d_l, K)$
- 21: **end if**
- 22: **end for**
- 23: **end for**
- 24: return $BP(s_l, d_l, K)$;

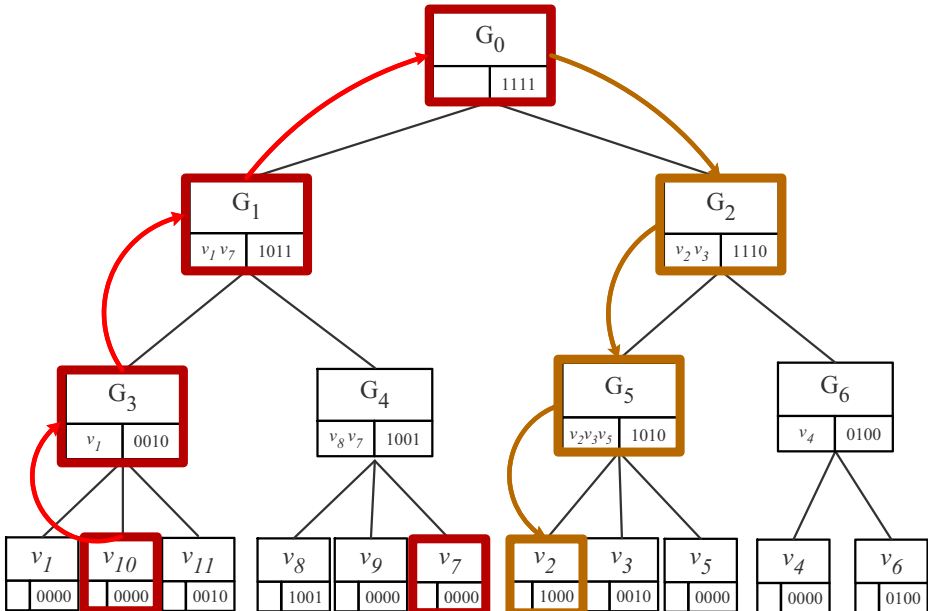


Figure 6 Path from v_{10} to nearest node with keyword *book*

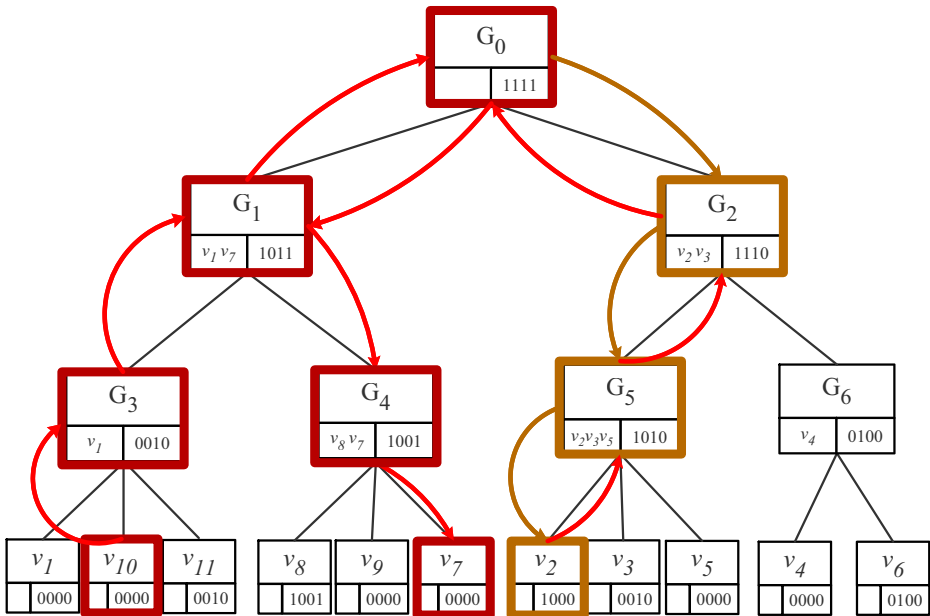


Figure 7 Path from v_{10} to v_7 passing through keyword *book*

As we have found the candidates for each keyword specified by the user, we can do best path search from s_l to the candidates, then to d_l . Q_k consists of v_2 and v_6 . So what we have to do is to find which vertex in the candidate queue Q_k is the nearest from s_l (v_{10}). In this case, v_2 is the nearest so we have to find the shortest path $\delta(v_{10}, v_2) = \{v_{10}, v_1, v_2\}$ with total distance $dist(v_{10}, v_2) = 5$. Next, we have to find the nearest next candidate from v_2 , which is v_6 . Then we establish another shortest path $\delta(v_2, v_6) = \{v_2, v_5, v_4, v_6\}$ with total distance $dist(v_2, v_6) = 7$. Since there is no more candidate in the queue, then it means that we have found all the keywords specified by the user in our path. Thus we can establish the final path from the last candidate to the destination d_l ($\delta(v_6, v_7) = \{v_6, v_4, v_5, v_3, v_7\}$ with total distance $dist(v_6, v_7) = 11$). The result of the best path $BP(v_{10}, v_7, \{book, cinema\}) = \{v_{10}, v_1, v_2, v_5, v_4, v_6, v_4, v_5, v_3, v_7\}$.

Based on our experiment, this algorithm runs faster than the baseline algorithm even though the approximation is not 100% accurate. However the approximation result is close to the baseline result even when the algorithm runs with a large dataset. The pseudo-code of this algorithm is given in Algorithm 3.

5.3 Ancestor priority approximation search

In this paper, we propose another approximation algorithm. This algorithm utilizes the common ancestor between the source and destination locations with the purpose of minimizing

the tree traversal time. Sometimes when we are trying to find one or more keywords in the *IG-Tree*, we have to travel through most of the tree nodes even though the source and destination locations are on the same partition node. However in this algorithm, the idea is to traverse only on the branch of an ancestor node. This is basically to do early pruning through the common ancestor between source and destination locations in *IG-Tree*.

Algorithm 3 Optimal Distance Approximation Search

Input: $IG - Tree, s_l, d_l, K$ in
Output: $BP(s_l, d_l, K)$ out

- 1: Index K to inverted file K_{if}
- 2: Find s_l on $IG - Tree$
- 3: Find d_l on $IG - Tree$
- 4: **for** each inverted keyword k_n in K_{if} **do**
- 5: **if** $k_n +$ **then**
- 6: Find shortest path $\delta(s_l, k_n)$
- 7: Find shortest path $\delta(k_n, d_l)$
- 8: $dist(s_l, d_l) = dist(s_l, k_n) + dist(k_n, d_l)$
- 9: **end if**
- 10: **if** current path is the shortest **then**
- 11: Store (k_n) location to queue Q_k
- 12: **end if**
- 13: **end for**
- 14: **for** Q_k is not empty **do**
- 15: **if** current path is 0 **then**
- 16: Find INN (s_l, Q_k)
- 17: **else**
- 18: Find INN $(Q_{kn} - 1, Q_k)$
- 19: **end if**
- 20: Find shortest path $\delta(s_l, Q_{kn})$
- 21: Remove Q_{kn} from queue Q_k
- 22: **end for**
- 23: Find shortest path $\delta(Q_{kn}, d_l)$

As an example, a user invokes a query with source location in vertex v_{10} , destination location in vertex v_7 , and the preferred keyword is *book*. The query keyword inverted index in this case is 1000 for keyword *book*. Looking at the *IG-Tree* in Figure 8, the common ancestor between v_{10} and v_7 is G_1 . Thus in this algorithm we are going to only focus on the branch under G_1 , especially if the user given keyword is available in this branch. As the query inverted index is 1000 and the inverted index attached in G_1 is 1011, we can see that the query keyword exists in G_1 , so we can definitely focus on this node to find the best path from v_{10} to v_7 while passing by a keyword *book*.

The way the Ancestor Priority Search algorithm works is similar to the Optimal Distance Approximation algorithm once we know which branch we need to work on. Firstly we

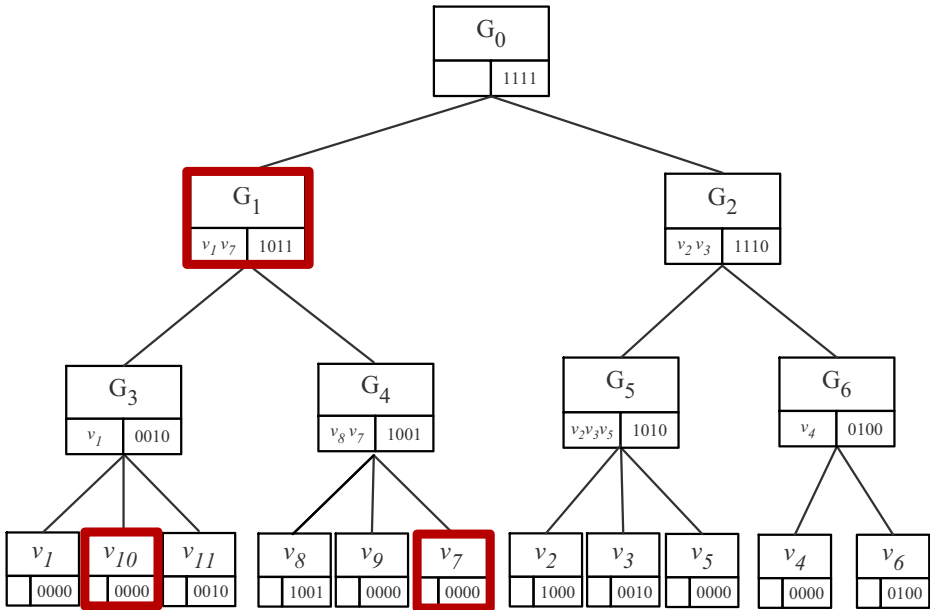


Figure 8 G_1 as the common ancestor of v_{10} and v_7

have to compute the best path of each keyword, then recording the candidate vertices into a queue Q_k in order to find the final multiple keywords best path query. Continuing from the previous example, the focus now is only on the branch of G_1 . So we do not need to travel to other branches outside partition G_1 . In this case, we have to find the best path for each query keyword first. But since there is only one keyword, then we can find the Best Path directly. The way we find each keyword is through the inverted index attached in the *IG-Tree* and traverse down until we found which vertex has the keyword. We know that G_1 has 1000 so we have to check its immediate children. G_3 does not have 1000, while G_4 has 1000, thus we need to traverse down the partition of G_4 in order to find the keyword. The children of G_4 are v_8 , v_9 , and v_7 . Only v_8 has 1000, therefore we have to find the best path from s_l to v_8 and then to v_7 as depicted in Figure 9. Similar to the previous algorithm, we have to find the shortest path $\delta(v_{10}, v_8)$ and $\delta(v_8, v_7)$ to help finding the best path. The shortest path $\delta(v_{10}, v_8) = \{v_{10}, v_1, v_8\}$, while the shortest path $\delta(v_8, v_7) = \{v_8, v_1, v_7\}$. As there is only one keyword, therefore the Best Path $BP(v_{10}, v_7, \{book\}) = \{v_{10}, v_1, v_8, v_1, v_7\}$.

The previous case however does not always happens because if the keyword does not exist in the current ancestor node, we have to go to its parent node and check whether the query keyword is available in the parent node. If it does not, then we have to keep going to the upper node until we can find the query keyword. Once the query keyword is found in the node, then we can continue the best path search. For example assume that a user wants to find the best path from v_{10} to v_7 while passing through a *cinema*. The inverted index for *cinema* is 0100, while the common ancestor of v_{10} and v_7 is G_1 . The partition G_1 does not

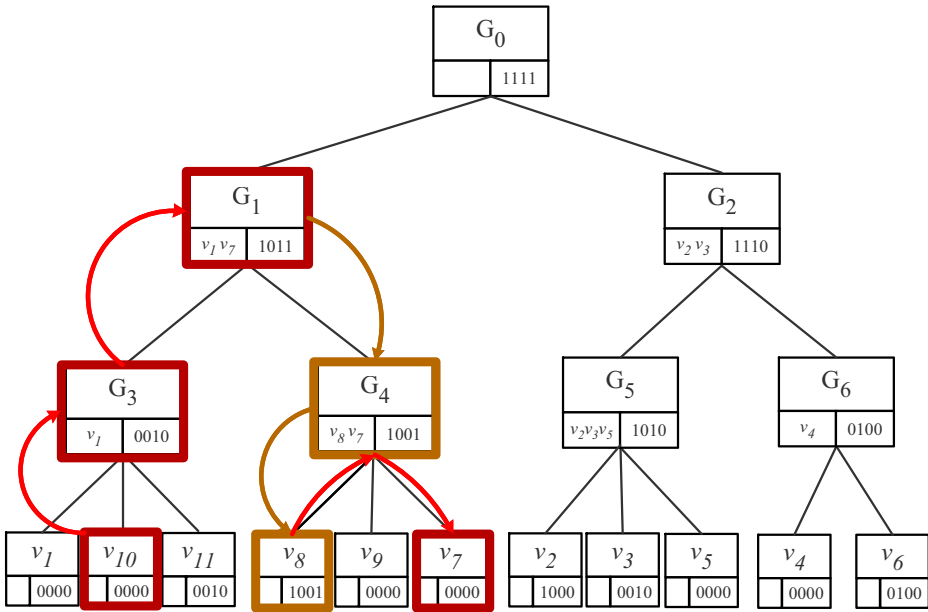


Figure 9 Path from v_{10} to v_7 passing through keyword *book*

have *cinema* in it since its inverted index is 1011, thus we have to find out whether *cinema* is available in G_1 's parent node. The parent node of G_1 is G_0 and its inverted index is 1111, which means that *cinema* exists in this partition. Therefore the best path search will cover the whole tree branches under G_0 .

The main advantage of this algorithm is in the early pruning. There is no need to explore the whole tree as we only need to focus on one branch through the common ancestor between the source and destination. However the disadvantage of this algorithm is that it has even lower accuracy compared to the Optimal Distance Approximation Search algorithm. Traversing under one branch does not guarantee the shortest path distance for best path since some keywords with closer distances might be located in other partitions. But this algorithm tries to compromise this with lesser tree traversal cost. The pseudo-code of this algorithm is given in Algorithm 4.

5.4 Euclidean-based approximation search

We propose another approximation algorithm in this paper. The idea behind this particular algorithm is to make use of the coordinate of each vertex and then find the best path through Euclidean distance before applying it into road network. This approximation algorithm is very fast compared to the previous algorithms since it is using Euclidean distance computation. However because of the usage of Euclidean distance on a road network data, the performance of this algorithm is quite low in terms of its accuracy.

Algorithm 4 Ancestor-Priority Search**Input:** $IG - Tree, s_l, d_l, K$ in**Output:** $BP(s_l, d_l, K)$ out

```

1: Index  $K$  to inverted file  $K_{if}$ 
2: Find  $s_l$  on  $IG - Tree$ 
3: Find  $d_l$  on  $IG - Tree$ 
4: Find common ancestor  $A$  of  $s_l$  and  $d_l$ 
5: Check if  $A$  contains all keywords in  $K_{if}$ 
6: while  $A \oplus K_{if}$  is 1 do
7:    $A = A$ 's parent node
8: end while
9: for each inverted keyword  $k_n$  in  $K_{if}$  do
10:  if  $k_n +$  then
11:    Find shortest path  $(s_l, k_n)$ 
12:    Find shortest path  $(k_n, d_l)$ 
13:  end if
14:  if current path is the shortest then
15:    Store  $(k_n)$  location to queue  $Q_k$ 
16:  end if
17: end for
18: for  $Q_k$  is not empty do
19:  if current path is 0 then
20:    Find INN  $(s_l, Q_k)$ 
21:  else
22:    Find INN  $(Q_{kn} - 1, Q_k)$ 
23:  end if
24:  Find shortest path  $(s_l, Q_{kn})$ 
25:  Remove  $Q_{kn}$  from queue  $Q_k$ 
26: end for
27: Find shortest path  $(Q_{kn}, d_l)$ 

```

The Euclidean-based Approximation has two main components, namely the Euclidean approximation (Algorithm 5 row 1-10) and the best road network path (Algorithm 5 row 11-20). In the Euclidean approximation part, we firstly need to find the Euclidean locations of both the source location s_l and the destination location d_l . Based on these two Euclidean locations, we calculate the best path in Euclidean distance for each keyword k_n in K_{if} . The way we find the best path for each keyword k_n is similar to the Optimal Distance Approximation algorithm, where we have to get the optimum shortest path of $\delta(s_l, k_n) + \delta(k_n, d_l)$ in Euclidean distance and then store k_n into a candidate queue Q_k to help establishing the final best path. Once we have found the best path of each keyword k_n , we move to the second part, which is the road network path.

In the road network path component of Euclidean-based Approximation algorithm, the best path search is done in a similar fashion as the previous algorithms where we have to find the nearest candidate Q_{k1} from s_l then establish the shortest path $\delta(s_l, Q_{k1})$ between s_l and the candidate Q_{k1} in road network distance instead of the Euclidean distance. After the shortest path $\delta(s_l, Q_{k1})$ is established, we need to find the next nearest candidate Q_{kn} from the Q_{k1} and establish the shortest path $\delta(Q_{k1}, Q_{kn})$. We repeat the same step until

every single candidate in the queue Q_k has been visited. Then we can find the shortest path $\delta(Q_{kn}, d_l)$ to end the trip.

Algorithm 5 Euclidean-based Approximation Search

Input: $IG - Tree, s_l, d_l, K$ in
Output: $BP(s_l, d_l, K)$ out

- 1: Index K to inverted file K_{if}
- 2: Find s_l 's Euclidean location on $IG - Tree$
- 3: Find d_l 's Euclidean location on $IG - Tree$
- 4: **for** each inverted keyword k_n in K_{if} **do**
- 5: **if** k_n+ **then**
- 6: Find Euclidean shortest path (s_l, k_n)
- 7: Find Euclidean shortest path (k_n, d_l)
- 8: **end if**
- 9: Store (k_n) location to queue Q_k
- 10: **end for**
- 11: **for** Q_k is not empty **do**
- 12: **if** current path is 0 **then**
- 13: Find INN (s_l, Q_k)
- 14: **else**
- 15: Find INN $(Q_{kn} - 1, Q_k)$
- 16: **end if**
- 17: Find Road Network shortest path (s_l, Q_{kn})
- 18: Remove Q_{kn} from queue Q_k
- 19: **end for**
- 20: Find Road Network shortest path (Q_{kn}, d_l)

The Euclidean distance is merely to help deciding which vertices to be visited based on the keywords chosen by the user. But at the end the best path's result is in road network distance. The approximation in this algorithm is very low as it can over-approximate the result up to 300% based on our experiment. However the running time of this algorithm is a lot faster compare to the other algorithms.

6 Experiment

In this section, we compare the efficiency and accuracy of the four Best Path query processing algorithms from Section 5: Baseline Algorithm (BruteForce), Optimal Distance Approximation Search (OptDist), Ancestor Priority Approximation Search (AncestorPriority), and Euclidean-based Approximation Search (Euclidean).

6.1 Settings

6.1.1 Environment

We perform our experiments on 2.5 GHz Intel Core i7-4870 CPU and 12 GB RAM running 64-bit Ubuntu. All of the algorithms were written in single-threaded C++.

Table 21 Road Network Datasets

Dataset	Description	# Vertices	# Edges
CAL	California	21,048	43,386
NY	New York City	264,346	733,846
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798

6.1.2 Datasets

We use real datasets from 9th DIMACS Implementation Challenge - Shortest Paths [20] and [55] for the road network datasets. We select four datasets: California, New York City, Colorado, and Florida. Table 21 provides the details of the size of the real-world road network datasets.

Meanwhile for the textual information we utilize keyword sets based on [55] and assign them into the vertices in the road network datasets. As the textual part needs to be able to detect whether the user gives one or more negative keywords, a number of negative sentiment analysis based words from [21, 22] are used in order to accommodate the negative keyword(s) in the user query.

6.1.3 Queries

In our experiments, we generate the keyword set K for the test queries with a random distribution from a keyword pool. The size of the keyword set in the test queries varies from 1 to 15 while the object density¹ of these keyword set varies from 1% to 30% of the whole road network datasets. We also evaluate the impact of varying the distance between s_l and d_l pairs, which are varied from 2% to 64% of the maximum distance between two vertices in the space.

6.2 Index evaluation

This section evaluates the proposed *IG-Tree* index for planning Best Path queries on road networks in terms of index building time and space consumption, and index reconstruction time(s) for negative keywords in the tested queries. Figure 10 shows the index building times and space consumption of the proposed *IG-Tree* and the *G-Tree* indices for New York City dataset. We see that index building time of *IG-Tree* is comparable to that of *G-Tree* though *IG-Tree* combines *IR²-Tree* with *G-Tree*. The index size of *IG-Tree* is slightly larger than that of *G-Tree* as inverted lists and Keyword Distance Matrices are maintained in *IG-Tree* in addition to Distance Matrices.

Finally, the times required to reconstruct the *IG-Tree* index for negative keywords in the Best Path queries are quite durable as we observe from Figure 11. The *IG-Tree* takes only ~ 0.8 secs to reconstruct the index for up to 10 (negative) keywords. However, we observe only a few keywords in K including negative keywords in route planning queries, which is around 5-6 keywords, in our usual life. Therefore, we believe that the index reconstruction

¹Object density: the quantity of keyword matched objects for each query keyword compared to the number of vertices in the road network.

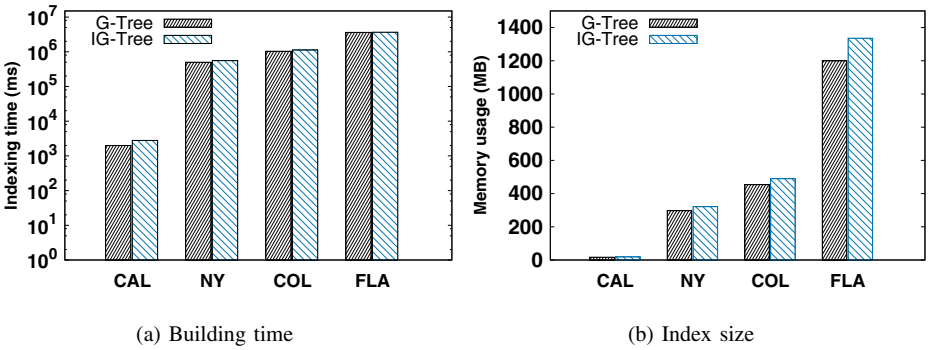


Figure 10 IG-Tree vs. G-Tree: index building time and size

time *IG-Tree* for few negative keywords would be pretty durable in practical applications of Best Path queries.

6.3 Performance study

We evaluate our query processing algorithms on two metrics, specifically on the running time and approximation accuracy. The approximation accuracy shows the percentage of the result accuracy produced by each algorithm compared to the expected correct result obtained by running the baseline algorithm.

6.3.1 Effect of k^+

The positive keywords (k^+) given by the users take a very important role in Best Path Query. Each k^+ must be visited at least once, therefore the more k^+ to be visited, it is expected that the running time also increases for every algorithm. Figure 12 shows the query performance as the number of k^+ increases. In these experiments, we specified the query keywords K to be all positive, without any negative keywords. The experiments show that the running times for the approximation algorithms (OptDist, AncestorPriority, Euclidean) run a lot better compared to the baseline algorithm. The baseline algorithm has the worst

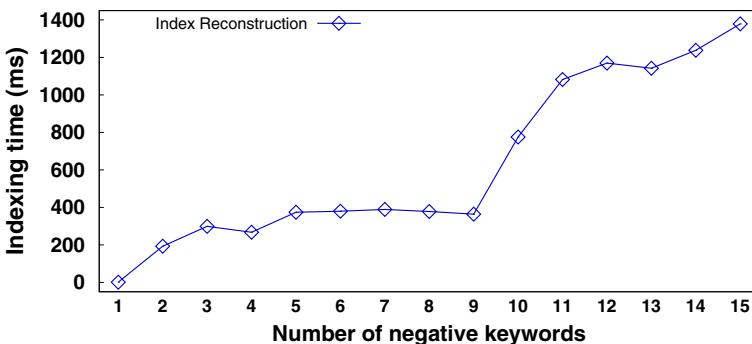


Figure 11 Index reconstruction times for varied number of negative keywords in K

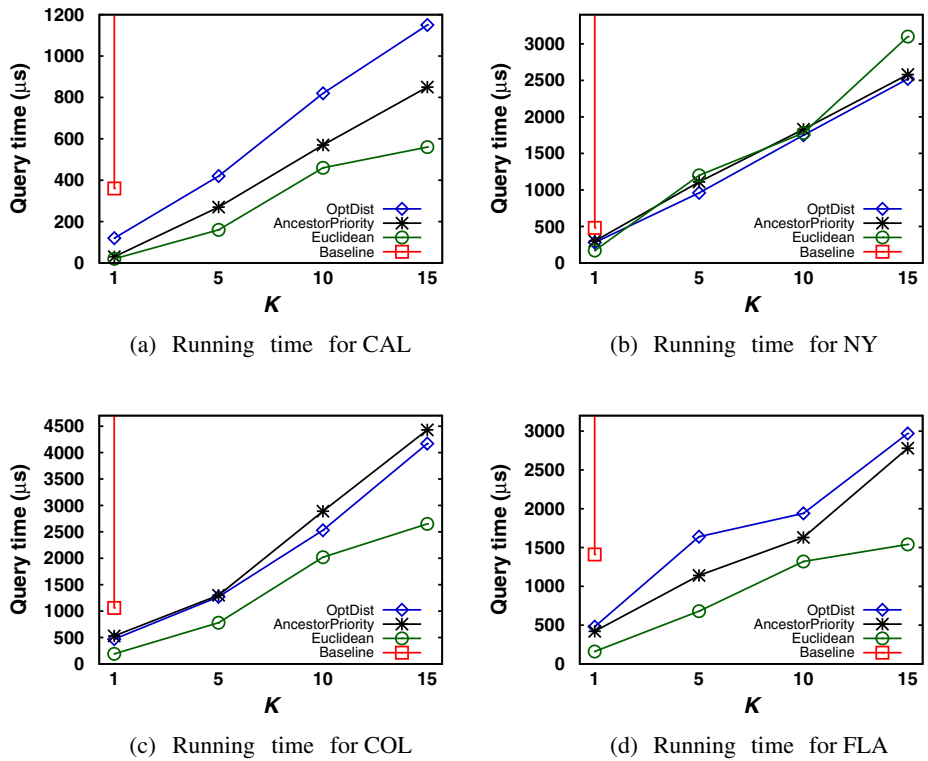


Figure 12 Query performance with all positive query keywords

running time among all as the time increases exponentially. According to our experiments, the average running time for baseline algorithm for $|K| = 1$ is 0.48 ms but it increases up to 21507.41 ms when $|K| = 5$. Even though baseline algorithm offers a precise solution, the amount of time taken to get the result is not suitable for daily usage. Imagine when we want to plan a trip to a new country and it takes 17 hours for us to get the best path with $|K| = 10$. This is definitely impossible to be used in everyday life.

As we proposed three approximation algorithms, we also evaluate the approximation accuracy for each algorithm. Figure 13 shows the percentage of accuracy of each approximation algorithm compared to the baseline algorithm. When K is only 1, all the three approximation algorithms have high accuracy. However when K increases, the accuracy decreases. The OptDist has the best approximation compared to the other two algorithms. Even though its approximation is not 100% accurate, the percentage of accuracy is still above 75%. It is very different from the Euclidean-based algorithm to which its approximation is very poor compared to others as the accuracy is only 6.83% when $K = 15$. The trend for Euclidean approximation algorithm is definitely the worst as the inaccuracy keeps escalating drastically.

Based on this experiment, we can see that OptDist performs better than the other three algorithms in terms of the running time. It also performs better than the other two approximation algorithms in terms of the approximation accuracy. So we can conclude that OptDist is the best choice when we want to invoke Best Path Query with various numbers of K .

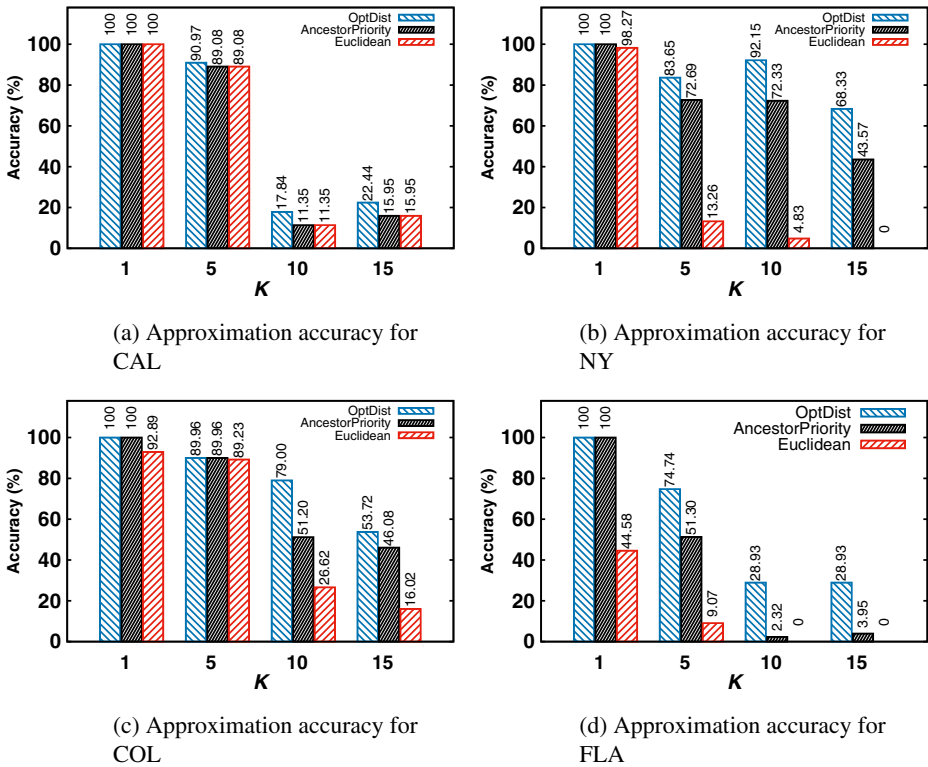


Figure 13 Approximation accuracy for all positive query keywords

6.3.2 Effect of k^-

The k^- has a great impact to the Best Path Query. As previously discussed in Section 4, there are several cases on what would happen to the *IG-Tree* when we found a k^- . A lot of times when k^- is located on the border, the path cannot be retrieved at all. So the distribution of k^- is always kept to be lesser than k^+ in this particular experiment to ensure that we can retrieve some results. For this experiment, we set the query keywords K to have both positive and negative keywords.

According to our experiment result in Figure 14, the Euclidean-based algorithm always has a faster running time compared to the other three algorithms. The OptDist and AncestorPriority are actually almost the same in terms of their running time even though the AncestorPriority still seems to be a bit faster than OptDist. Meanwhile the baseline algorithm has the worst running time as expected. Having a negative keyword k^- definitely affects the running time of some queries as there might be path reconstruction happening throughout the query processing. This also explains the difference between the time in Figures 12 and 14, where the running time in Figure 12 with all positive keywords does not require any path reconstruction so it is faster than the experiment result in Figure 14.

Another metric that we test is the accuracy of the approximation algorithms. Figure 15 shows the result of the approximation accuracy of each algorithm towards the result of baseline algorithm. The trend in Figure 15 is almost similar to the trend in Figure 13, which

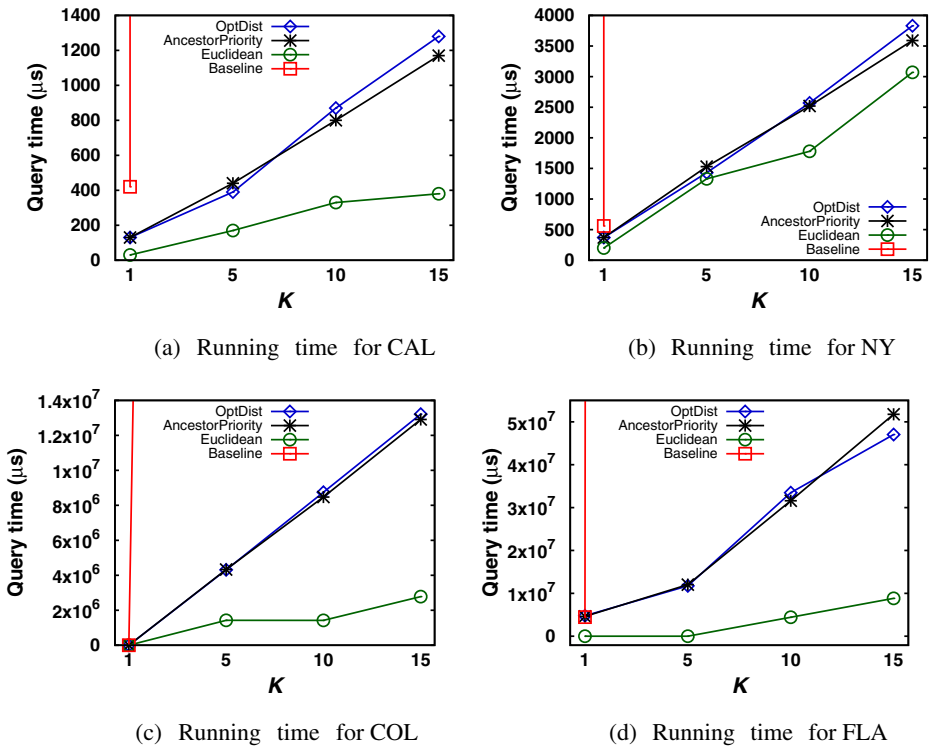


Figure 14 Query performance with combination of positive and negative query keywords

might indicate that the path reconstructions happening in these queries because of k^- does not truly have impact towards the accuracy. We can also see that the accuracy percentage of OptDist and AncestorPriority are the same for this case and both have better accuracy than the Euclidean-based algorithm. The Euclidean approximation is still the worst among the other algorithms with very low accuracy even though the running time is a bit faster than the others.

6.3.3 Effect of keyword densities

In this experiment evaluation, we want to observe the query performance when we increase the keyword densities. Figure 16 shows the running time for query within the density of 0.01 to 0.30. The running time of the baseline algorithm increases drastically compared to the rest. Even in the lower density case, specifically on 0.01 density, the baseline algorithm takes about 17x more time than the OptDist algorithm. The Euclidean-based algorithm however performs in a constant manner even though the density increases. The OptDist and AncestorPriority on the other hand have similar running time.

We also run an experiment on the running time when K is varied from 1 to 15 while the density for each keyword is 0.05. Figure 17 shows the result of this specific experiment where it interestingly shows that the constant running time for Euclidean algorithm. The trend indicates that Euclidean-based algorithm has the most constant running time especially for denser datasets. This is totally different from the other three algorithms, which

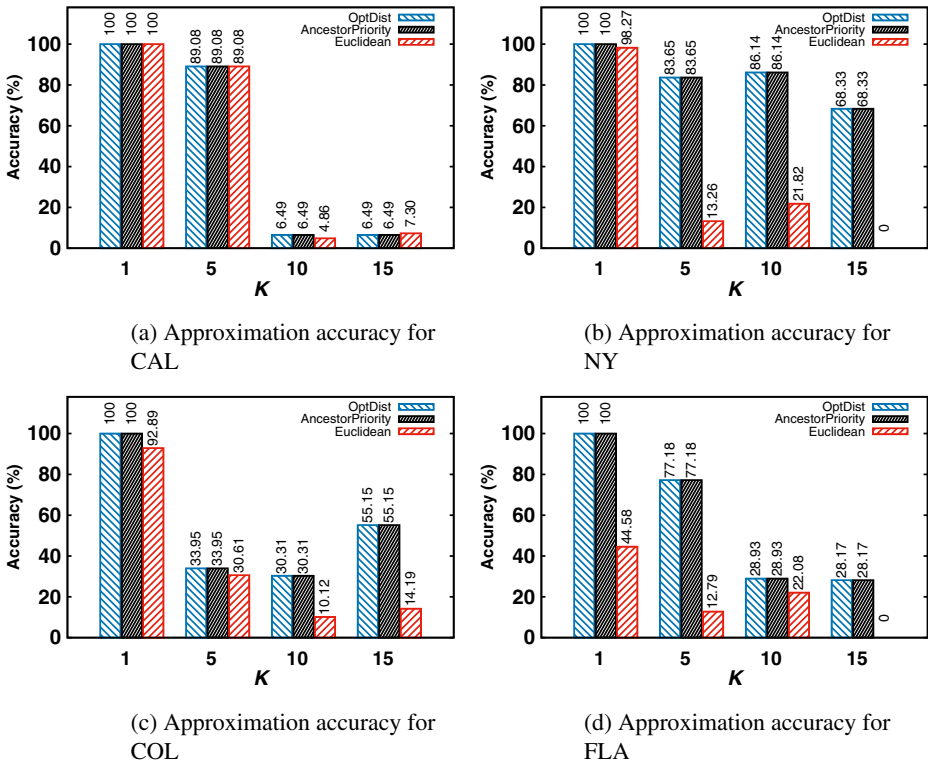


Figure 15 Approximation accuracy for datasets with negative keywords

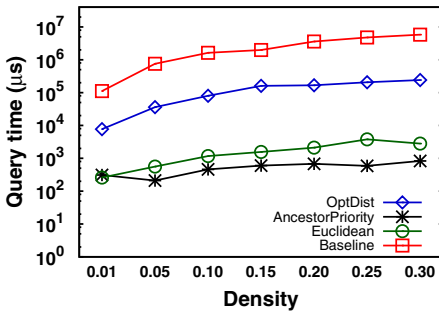
keep increasing with the increase in density. Even though Figures 12 and 14 run similar experiments, the results are different. In Figures 12 and 14, the keyword density is randomized and usually below 0.05.

We conducted another experiment to test the running time of query with all negative keywords while increasing the density of the keywords from 0.01 to 0.055. We only increase until 0.055 by the reason of having higher density of negative keywords will return no path/no result at all. Figure 18 shows the result of this particular experiment. Surprisingly, the running time of Euclidean algorithm drastically increases almost in the same trend as the baseline algorithm. Meanwhile, the OptDist and AncestorPriority are running in constant time manner when the keywords are all negative.

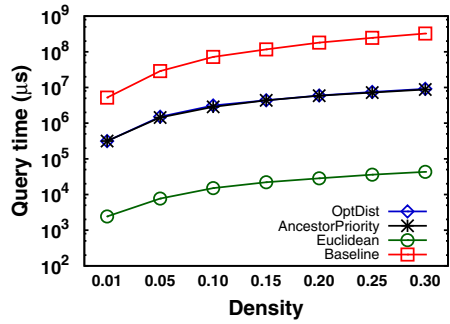
6.3.4 Effect of positive and negative keywords ratio

Figure 19 shows the running time on the positive and negative keywords ratio. The ratio is based on 0.01 density. In this experiment, we limit the ratio of the keywords (positive:negative) into 1:0, 0:1, 1:1, 5:0, 0:5, and 5:1. We exclude the ratio with negative keywords higher than the positive keywords as there is no path retrieved most of the time with this kind of query.

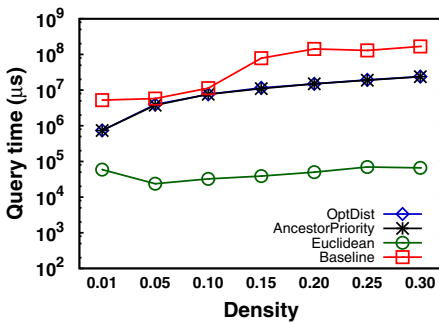
In Figure 19, we can see that the running time of OptDist algorithm increases if the number of query keywords increase (both positive and negative). The AncestorPriority has



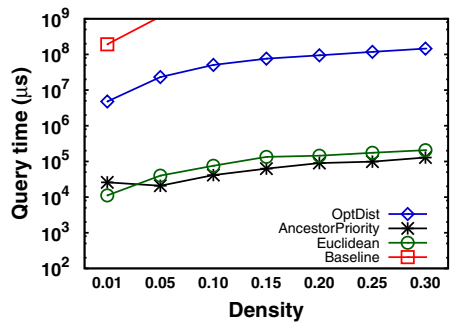
(a) Running time for CAL



(b) Running time for NY



(c) Running time for COL



(d) Running time for FLA

Figure 16 Query performance based on keyword density

similar trend with OptDist with the increase of time towards the more keywords involved. The increase in trend also happens to the baseline algorithm but it increases exponentially as what we have expected from the previous experiments. Nevertheless, the increasing trend does not happen to the Euclidean algorithm as it is comparatively constant in running time even with more positive keywords added into the query.

6.3.5 Effect of distance between s_l and d_l

We also evaluate the effect of varying the distance between s_l and d_l pairs. The distance between s_l and d_l are varied from 2%, 4%, 8%, 16%, 32%, up to 64% of the maximum distance between two points in the road network datasets. We set $|K| = 15$ by default and contain both positive and negative keywords. Figure 20 shows the experimental results of the source–destination distance. We do not include the baseline in Figure 20 since the runtime for 2% in NY dataset already reaches above 10,000 ms, which made a huge difference with the other three algorithms.

From the experimental result, we can see that the three algorithms are running in constant time even though the source-destination distance increases. Both OptDist and AncestorPriority have a similar trend, while Euclidean has better running time most of the time than the rest.

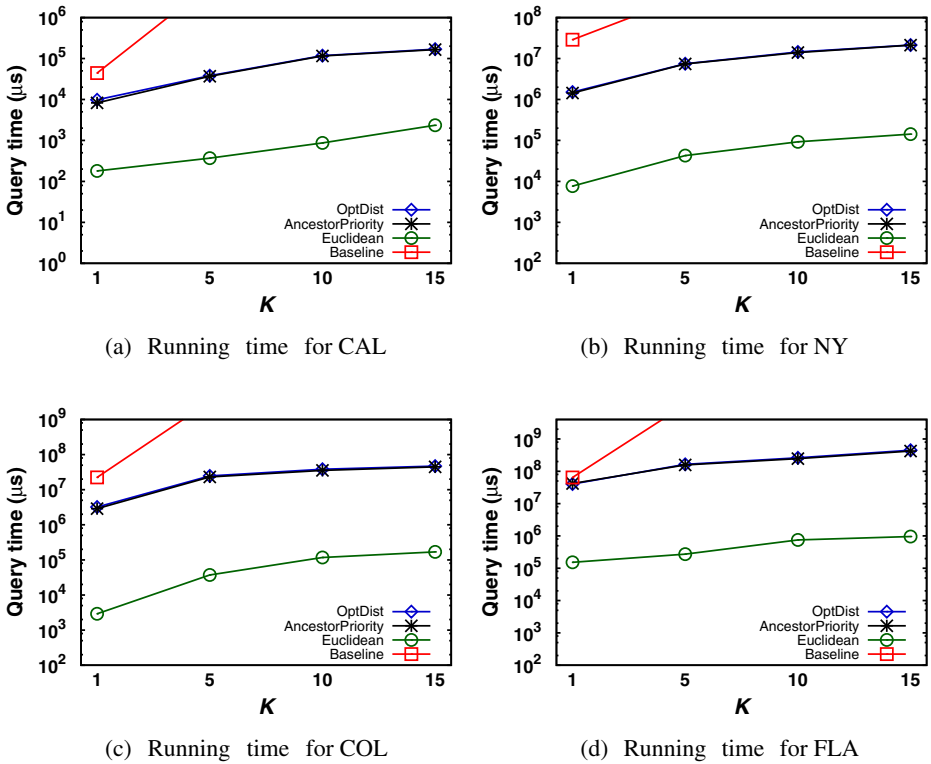


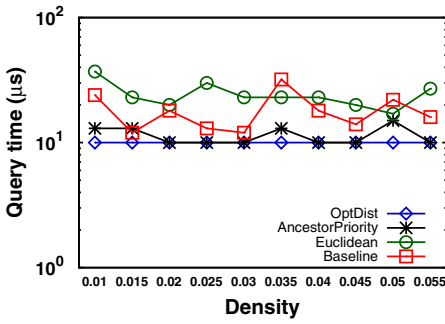
Figure 17 Query performance when K is varied (keyword density=0.05)

6.3.6 Summary

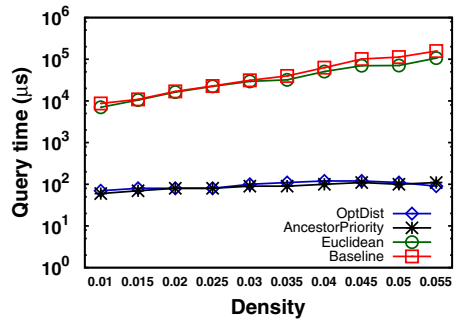
Based on our experimental study, each algorithm has its own strength. The baseline algorithm certainly offers accurate result, but it has the worst running time as it increases exponentially when the queries and dataset increases. The OptDist itself has the best approximation compared to the other two approximation algorithms (AncestorPriority and Euclidean). Its running time is definitely a lot better than the baseline algorithm but the AncestorPriority and Euclidean-based algorithms still beats it by a few microseconds, especially when all the keywords in the user query are positive.

The AncestorPriority often follows the trend of OptDist on both the running time and accuracy. In terms of running time, AncestorPriority is frequently faster by only a few microseconds compared to OptDist. On the other hand, the accuracy of AncestorPriority is slightly lower than OptDist because of the early pruning. Compared to the Euclidean, AncestorPriority still has better accuracy even though its speed is still slower than the Euclidean.

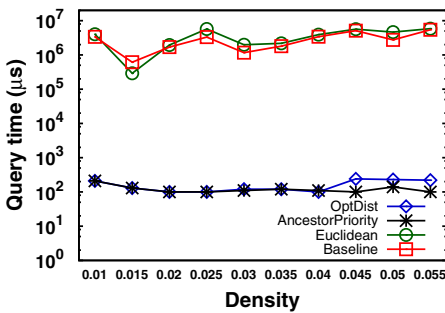
The Euclidean-based algorithm’s main strength is in its fast runtime. For a quick approximation, the Euclidean-based algorithm can be used but it has the lowest accuracy compared to the other algorithms. The Euclidean however does not perform well when the density of the negative keywords are high.



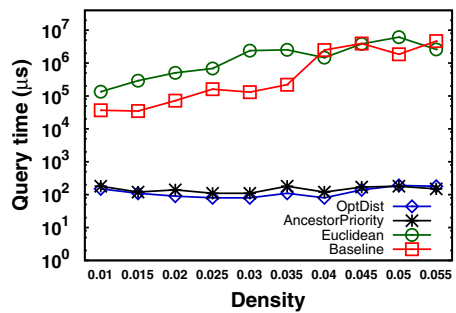
(a) Running time for CAL



(b) Running time for NY



(c) Running time for COL



(d) Running time for FLA

Figure 18 Query performance based on keyword density with all negative keywords

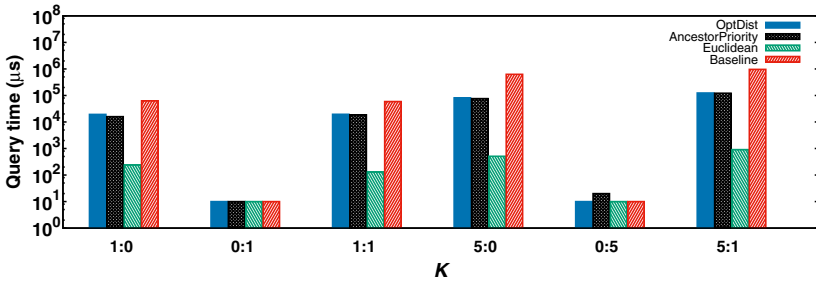
In general, OptDist offers the best solution in comparison with the other algorithms. Even though the Euclidean-based algorithm outperforms the running time of OptDist when the queries contain all positive keywords, OptDist still provide better approximation. OptDist is more stable in both its speed and approximation accuracy.

7 Related works

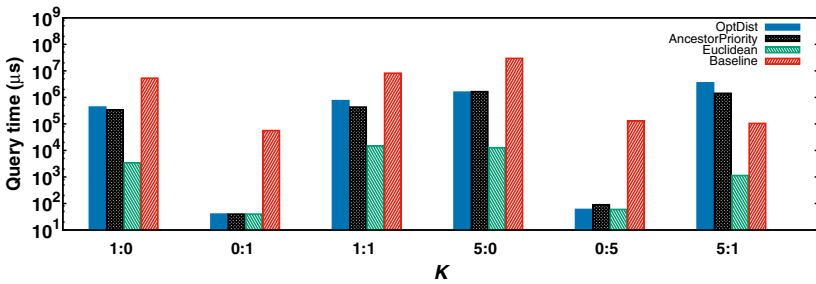
This section discusses the related works, specifically on Spatial Keyword and Route Planning Queries.

7.1 Spatial keywords queries

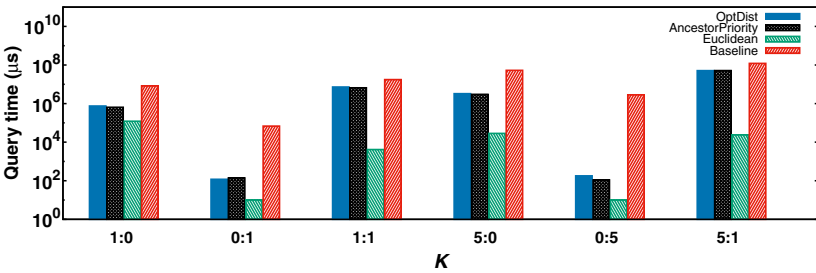
With the growth of geographical search engine, Spatial Keyword Queries becoming more crucial and popular among researchers. Most of the early works on Spatial Keyword Queries focus on queries like top- k nearest neighbor queries ($TkNN$) [7–9, 11, 15, 35, 42]. In $TkNN$ queries, the goal is to rank objects, measured the keywords similarity (between the object’s keyword and query) and the distance from the specified query location, in order to retrieve k



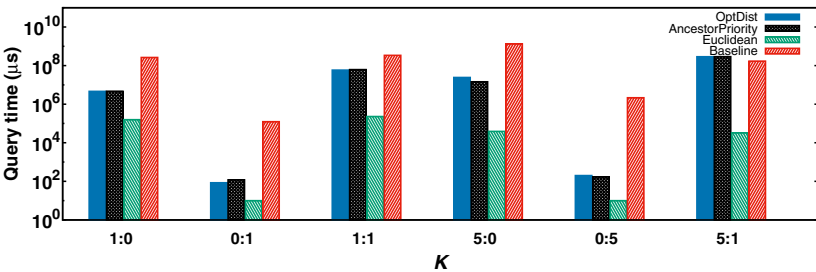
(a) Running time for CAL



(b) Running time for NY



(c) Running time for COL



(d) Running time for FLA

Figure 19 Running time based on keyword ratio (positive:negative)

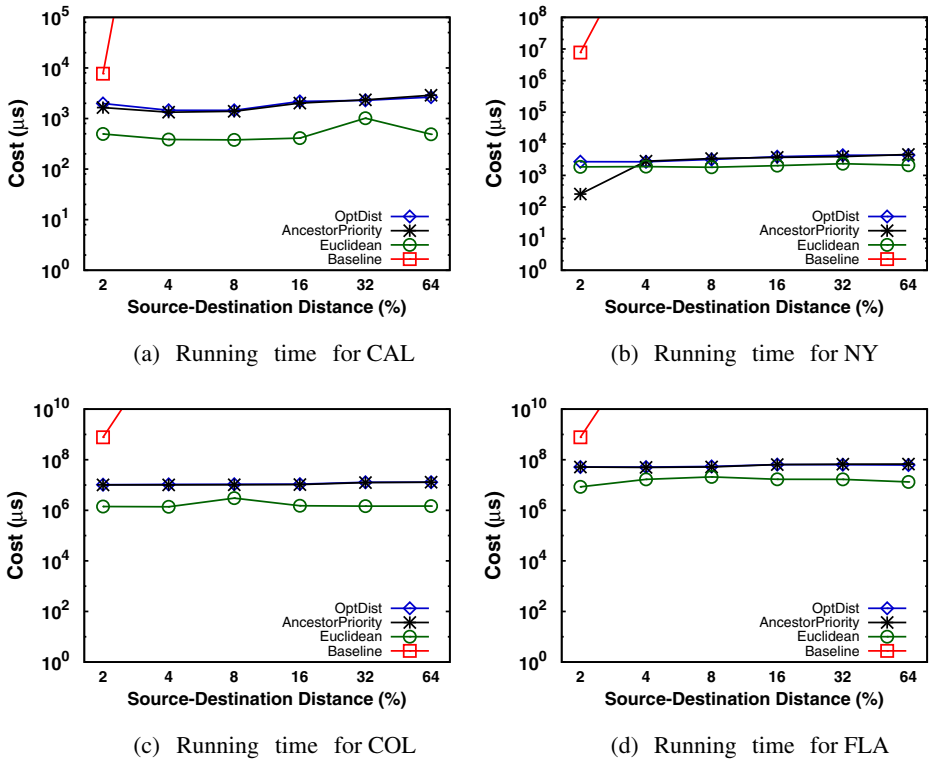


Figure 20 Effect of varying distance between s_i and d_i

number of objects with the highest ranking. As discussed earlier, this type of query mainly accepts user’s spatial location and keywords as input, and produces spatial objects with matching keywords as the output.

A lot of other Spatial Keyword Queries variants are also based on $TkNN$ queries, in which the works on these variants try to improve the $TkNN$ queries to be able to process moving objects [40], continuous objects [17], reverse top- k query [16, 32], joint queries [23, 41], or interactive $TkNN$ queries [49]. Besides the works on Spatial Keyword Queries that focus on $TkNN$ queries, some variants of Spatial Keyword Queries have also been proposed, such as the collective Spatial Keyword querying [10, 31, 48], diversified Spatial Keyword search [47], region-based query [13], scalable continual top- k query [43], reverse spatial and textual k nearest neighbor query [34], spatio-textual data clustering [14], fuzzy keyword search [2], and m -closest keyword queries [46]. However, none of these queries can be classified as route planning queries.

As Spatial Keyword Queries become varied, a number of indexing techniques that are able to process both spatial and textual data have been proposed in the past years. A lot of the indexing technique on Euclidean space are utilizing the R-Tree in which they attach additional textual information into the R-Tree to be capable of computing textual data. Some of those R-Tree based indices are IR-Tree [28], bR^* -Tree [45], and IR^2 -Tree [15]. The IR^2 -Tree itself have been discussed in Section 4 as it inspired us to develop the IG -Tree for planning Best Path queries.

Several indexing techniques for Spatial Keyword Queries on road networks have also been studied recently. These indexing techniques are a lot more complex compared to indices for Euclidean data spaces. One of the earlier work was proposed by Rocha-Junior et al. [35] where their basic indexing architecture consists of four components. The first component is spatial component which is using the network R-Tree. The second is adjacency component, which it uses adjacency B-Tree to traverse the network. The third component is mapping component, which it uses Map B-Tree to map the adjacency edges with MBR that encloses the edges. The last component is the spatio-textual component, which it stores the spatial and textual properties of the objects. Another work on road networks is also done by Luo et al. [33]. They introduced a new indexing technique that is very different from Rocha-Junior et al. [35]. The proposed index, which is the Node-Partition-Distance (NPD) index, keeps useful distances so that the exact distance and the query keyword coverage can be computed independently. Another recent work on spatial keywords index on road networks is also proposed by Li et al. [30]. They proposed SKQAI, a novel air index for spatial keyword query processing on road networks. The SKQAI indexing technique consists of three components: weighted Quad-Tree of road network, keyword Quad-Trees, and network distance bound array.

We see that researchers have proposed many different indexing techniques in order to process diverse spatial keyword queries efficiently through the past few years. But the indexing techniques proposed still treat the spatial data part and the textual part as two different entities. Current techniques often adopts hybrid indexing, in which they have separate indices for the spatial data and the textual data and then combines both indices, especially for road networks. Looking at the existing studies, we find that these indices are not applicable to route planning queries in Spatio-Textual data, e.g., Best Path queries.

7.2 Route planning queries

There are a number of similar route planning queries as Best Path. The most well known query is the Trip Planning Route Query (TPQ), which retrieves the best trip from two different locations that passes at least one point from each of the chosen categories [27]. Ever since the invention of TPQ, many new studies start to investigate the variation and application of TPQ in certain areas, such as Group TPQ (GTP) [19] for processing multiple users' trip, and a recent study on TPQ with Location Privacy [37] in order to protect user's location privacy. Another popular route planning query is Optimal Sequenced Route Queries (OSR), a spatial query that finds the minimum route distance from a source location and passing through a set of sequenced categories [36]. However, these queries do not consider any keywords processing. Best Path Query focus on Spatio-Textual field, which in this case it needs to process the textual part of the objects. Another difference is that all of these existing works do not take into consideration any negative keywords. Everything in the chosen categories in TPQ and OSR must be visited, while in Best Path there are some categories that we have to avoid which increases the complexity of the problem.

In Spatio-Textual area itself, there is one study on route planning to the best of our knowledge, which is the Keyword-aware Optimal Route Search (KOR) [12]. It is a query that finds an optimal route that covers a set of user given keywords with a specific budget constraints and objective score [12]. In conjunction to Best Path Query, KOR is different as the query requires budget constraint for processing. KOR also does not take into consideration negative keywords as what Best Path does.

Most of the route planning problems is regarded as a generalization of Traveling Salesman Problem (TSP) problem [3, 4]. They are *NP*-hard problems. The solutions offered

for these queries are polynomial time approximation algorithms. Even so, the solutions offered do not involve significant pre-processing. This causes more computation, particularly since the computation requires the processing of both spatial and textual relevancy. So having an on-the-go solution does not always guarantee performance efficiency, especially on Spatio-Textual field. Therefore in this research we attempt to provide a better solution to this problem by offering a pre-processing index that incorporate both Keywords and Spatial Road Networks.

8 Conclusion

In this paper, we introduce a new variant of Spatial Keywords Query, which is the Best Path Query. The Best Path Query is an *NP-Hard* problem as it can be reduced to the Travelling Salesman Problem (TSP). Throughout our study, we develop an indexing technique called the *IG-Tree* that can process both spatial and textual information. This indexing technique can be used on various types of Spatial Keywords Query, especially on the Best Path Query. Three algorithms to solve the Best Path Query are also proposed in this paper, namely the Optimal Distance Approximation Search, Ancestor Priority Search, and the Euclidean-based Approximation solution. Each algorithm has its own strengths and weaknesses. The effectiveness and efficiency of the proposed algorithms are demonstrated through our extensive experiments. As a possible future work, we can further improve the *IG-Tree* to include keyword scoring function in order to increase the keyword search accuracy.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

1. Adhinugraha, K.M., Taniar, D., Indrawan, M.: Finding reverse nearest neighbors by region. *Concurrency Comput. Pract. Exp.* **26**(5), 1142–1156 (2014)
2. Alsubaiee, S., Li, C.: Fuzzy keyword search on spatial data. In: International Conference on Database Systems for Advanced Applications, pp. 464–467 (2010)
3. Arora, S.: Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM* **45**(5), 753–782 (1998)
4. Arora, S.: Approximation schemes for np-hard geometric optimization problems: a survey. *Math. Program.* **97**(1), 43–69 (2003)
5. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-Tree: an efficient and robust access method for points and rectangles. In: ACM SIGMOD, pp. 322–331 (1990)
6. Chen, Y.Y., Suel, T., Markowetz, A.: Efficient query processing in geographic Web search engines. In: ACM SIGMOD, pp. 277–288 (2006)
7. Chen, L., Cong, G., Jensen, C.S., Wu, D.: Spatial keyword query processing: an experimental evaluation. In: Proceedings of the VLDB Endowment, vol. 6, pp. 217–228 (2013)
8. Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial Web objects. *Proc. VLDB Endow.* **2**(1), 337–348 (2009)
9. Cao, X., Cong, G., Jensen, C.S.: Retrieving top-k prestige-based relevant spatial Web objects. *Proc. VLDB Endow.* **3**(1-2), 373–384 (2010)
10. Cao, X., Cong, G., Jensen, C.S., Ooi, B.C.: Collective spatial keyword querying. In: ACM SIGMOD, pp. 373–384 (2011)
11. Cao, X., Chen, L., Cong, G., Jensen, C.S., Qu, Q., Skovsgaard, A., Wu, D., Yiu, M.L.: Spatial keyword querying. In: Conceptual Modeling, pp. 16–29 (2012)
12. Cao, X., Chen, L., Cong, G., Guan, J., Phan, N.T., Xiao, X.: Kors: keyword-aware optimal route search system. In: IEEE ICDE, pp. 1340–1343 (2013)

13. Cao, X., Cong, G., Jensen, C.S., Yiu, M.L.: Retrieving regions of interest for user exploration. *Proc. VLDB Endow.* **7**(9), 733–744 (2014)
14. Choi, D.W., Chung, C.W.: A k-partitioning algorithm for clustering large-scale spatio-textual data. *Inf. Syst.* **64**(Supplement C), 1–11 (2017)
15. De Felipe, I., Hristidis, V., Rishe, N.: Keyword search on spatial databases. In: *IEEE ICDE*, pp. 656–665 (2008)
16. Gao, Y., Qin, X., Zheng, B., Chen, G.: Efficient reverse top-k boolean spatial keyword queries on road networks. *IEEE Trans. Knowl. Data Eng.* **27**(5), 1205–1218 (2015)
17. Guo, L., Shao, J., Aung, H., Tan, K.L.: Efficient continuous top-k spatial keyword queries on road networks. *GeoInformatica* **19**(1), 29–60 (2015)
18. Hariharan, R., Hore, B., Li, C., Mehrotra, S.: Processing spatial-keyword (Sk) queries in geographic information retrieval (Gir) systems. In: *ACM SSDBM*, pp. 16–16 (2007)
19. Hashem, T., Hashem, T., Ali, M.E., Kulik, L.: Group trip planning queries in spatial databases. In: *SSTD*, pp. 259–276 (2013)
20. <http://www.dis.uniroma1.it/challenge9/download.shtml>. Last accessed 22 January 2018
21. <http://www.wjh.harvard.edu/~inquirer/No.html>. Last accessed 22 January 2018
22. <https://github.com/jeffreymbreen/twitter-sentiment-analysis-tutorial-201107/blob/master/data/opinion-lexicon-English/negative-words.txt>. Last accessed 22 January 2018
23. Hu, H., Li, G., Bao, Z., Feng, J., Wu, Y., Gong, Z., Xu, Y.: Top-k spatio-textual similarity join. *IEEE Trans. Knowl. Data Eng.* **28**(2), 551–565 (2016)
24. Hwang, K., Cho, S.: A lifelog browser for visualization and search of mobile everyday-life. *Mob. Inf. Syst.* **10**(3), 243–258 (2014)
25. Jones, C.B., Abdelmoty, A.I., Finch, D., Fu, G., Vaid, S.: *Geographic information science: proceedings of the third international conference, GIScience*, Chap. The spirit spatial search engine: architecture, ontologies and spatial indexing (2004)
26. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: *Proceedings of the ACM/IEEE Conference on Supercomputing* (1995)
27. Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., Teng, S.H.: On trip planning queries in spatial databases. In: *SSTD*, pp. 273–290 (2005)
28. Li, Z., Lee, K.C.K., Zheng, B., Lee, W.C., Lee, D., Wang, X.: Ir-tree: an efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.* **23**(4), 585–599 (2011)
29. Li, Y., Wu, D., Xu, J., Choi, B., Su, W.: Spatial-aware interest group queries in location-based social networks. *Data Knowl. Eng.* **92**(Supplement C), 20–38 (2014)
30. Li, Y., Li, G., Li, J., Yao, K.: Skqai: a novel air index for spatial keyword query processing in road networks. *Inf. Sci.* **430–431**(Supplement C), 17–38 (2018)
31. Long, C., Wong, R.C.W., Wang, K., Fu, A.W.C.: Collective spatial keyword queries: a distance owner-driven approach. In: *ACM SIGMOD*, pp. 689–700 (2013)
32. Lu, J., Lu, Y., Cong, G.: Reverse spatial and textual k nearest neighbor search. In: *ACM SIGMOD*, pp. 349–360 (2011)
33. Luo, S., Luo, Y., Zhou, S., Cong, G., Guan, J., Yong, Z.: Distributed spatial keyword querying on road networks. In: *EDBT*, pp. 235–246 (2014)
34. Luo, C., Junlin, L., Li, G., Wei, W., Li, Y., Li, J.: Efficient reverse spatial and textual k nearest neighbor queries on road networks. *Knowl-Based Syst.* **93**(Supplement C), 121–134 (2016)
35. Rocha-Junior, J.B., Nørnvåg, K.: Top-K spatial keyword queries on road networks. In: *EDBT*, pp. 168–179 (2012)
36. Sharifzadeh, M., Kolahdouzan, M., Shahabi, C.: The optimal sequenced route query. *VLDB J.* **17**(4), 765–787 (2008)
37. Soma, S.C., Hashem, T., Cheema, M.A., Samrose, S.: Trip planning queries with location privacy in spatial databases. *World Wide Web* **20**(2), 205–236 (2017)
38. Waluyo, A.B., Srinivasan, B., Taniar, D.: Research in mobile database query optimization and processing. *Mob. Inf. Syst.* **1**(4), 225–252 (2005)
39. Waluyo, A.B., Taniar, D., Rahayu, W., Srinivasan, B.: Mobile service oriented architectures for n-queries. *J. Netw. Comput. Appl.* **32**(2), 434–447 (2009)
40. Wu, D., Yiu, M.L., Jensen, C.S., Cong, G.: Efficient continuously moving top-k spatial keyword query processing. In: *IEEE ICDE*, pp. 541–552 (2011)
41. Wu, D., Yiu, M.L., Cong, G., Jensen, C.S.: Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.* **24**(10), 1889–1903 (2012)
42. Xu, J., Lu, H.: Efficiently answer top-k queries on typed intervals. *Inf. Syst.* **71**(Supplement C), 164–181 (2017)

43. Xu, Y., Guan, J., Li, F., Zhou, S.: Scalable continual top-k keyword search in relational databases. *Data Knowl. Eng.* **86**, 206–223 (2013)
44. Yairi, I., Igi, S.: Mobility support gis with universal-designed data of barrier/barrier-free terrains and facilities for all pedestrians including the elderly and the disabled. In: *IEEE International Conference on Systems, Man and Cybernetics*, vol. 4, pp. 2909–2914 (2006)
45. Zhang, D., Chee, Y.M., Mondal, A., Tung, A.K., Kitsuregawa, M.: Keyword search in spatial databases: towards searching by document. In: *IEEE ICDE*, pp. 688–699 (2009)
46. Zhang, D., Ooi, B.C., Tung, A.K.H.: Locating mapped resources in Web 2.0. In: *IEEE ICDE*, pp. 521–532 (2010)
47. Zhang, C., Zhang, Y., Zhang, W., Lin, X., Cheema, M.A., Wang, X.: Diversified spatial keyword search on road networks. In: *EDBT*, pp. 367–378 (2014)
48. Zhang, P., Lin, H., Yao, B., Lu, D.: Level-aware collective spatial keyword queries. *Inf. Sci.* **378**(Supplement C), 194–214 (2017)
49. Zheng, K., Su, H., Zheng, B., Shang, S., Xu, J., Liu, J., Zhou, X.: Interactive top-k spatial keyword queries. In: *IEEE ICDE*, pp. 423–434 (2015)
50. Zhong, R., Fan, J., Li, G., Tan, K.L., Zhou, L.: Location-aware instant search. In: *ACM CIKM*, pp. 385–394 (2012)
51. Zhong, R., Li, G., Tan, K.L., Zhou, L.: G-Tree: an efficient index for knn search on road networks. In: *ACM CIKM*, pp. 39–48 (2013)
52. Zhong, R., Li, G., Tan, K.L., Zhou, L., Gong, Z.: G-tree: an efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.* **27**(8), 2175–2189 (2015)
53. <http://www.statisticbrain.com/mobile-browser-vs-application-preferences/>
54. <http://blog.globalwebindex.net/chart-of-the-day/top-global-smartphone-apps-who-s-in-the-top-10/>
55. <http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm>