An Index Method for the Shortest Path Query on Vertex Subset for the Large Graphs

Zian Pan, Yajun Yang(✉), and Qinghua Hu

College of Intelligence and Computing, Tianjin University, Tianjin, China
{panzian,yjyang,huqinghua}@tju.edu.cn

Abstract. Shortest path query is an important problem in graphs and has been well-studied. In this paper, we study a special kind of shortest path query on a vertex subset. Most of the existing works propose various index techniques to facilitate shortest path query. However, these indexes are constructed for the entire graphs, and they cannot be used for the shortest path query on a vertex subset. In this paper, we propose a novel index named pb-tree to organize various vertex subsets in a binary tree shape such that the descendant nodes on the same level of pb-tree consist of a partition of their common ancestors. We further introduce how to calculate the shortest path by pb-tree. The experimental results on three real-life datasets validate the efficiency of our method.

Keywords: Shortest path · Vertex subset · Index

1 Introduction

Graph is an important data model to describe the relationships among various entities in the real world. The shortest path query is a fundamental problem on graphs and has been well studied in the past couple of decades. In this paper, we study a special case of the shortest path query problem. Consider the following applications in the real world. In social networks, some users need to investigate the shortest path inside a specified community for two individuals. For example, someone intends to know another by the peoples with the same hobby or occupation. In transportation networks, some vehicles are restricted to a designated area such that they need to know the shortest route inside such area. The query problem in the above applications can be modeled as the shortest path query on a given vertex set for graphs. Given a graph $G(V, E)$ and a vertex subset $V_s \in V$, it is to find the shortest path from the starting vertex v_s to the ending vertex v_e on the induced subgraph of G on V_s.

It is obvious that the shortest path on a vertex subset V_s can be searched by the existing shortest path algorithms, e.g. Dijkstra algorithm. However, these algorithms are not efficient for the shortest path query on the large graphs. Most existing works propose various index techniques to enhance the efficiency of the

X. Wang et al. (Eds.): APWeb-WAIM 2020, LNCS 12317, pp. 69–85, 2020.
https://doi.org/10.1007/978-3-030-60259-8_6

shortest path query on the large graphs. The main idea of these works is that: build an index to maintain the shortest paths for some pairs of vertices in a graph. Given a query, algorithms first retrieve the shortest path to be visited among the vertices in the index and then concatenate them by the shortest paths which are not in the index. Unfortunately, such index techniques cannot be used for the shortest path problem proposed in this paper. It is because the vertex subset V_s is "dynamic". Different users may concern about the shortest path on distinct vertex subset V_s. The indexes for the entire graph G may not be suitable for the induced graph G_s on some given vertex subset V_s. The shortest path searched using the indexes for entire graph G may contain some vertices that are not in V_s. Therefore, the important issue is to develop an index technique such that it can be utilized for answering the shortest path query on various vertex subsets.

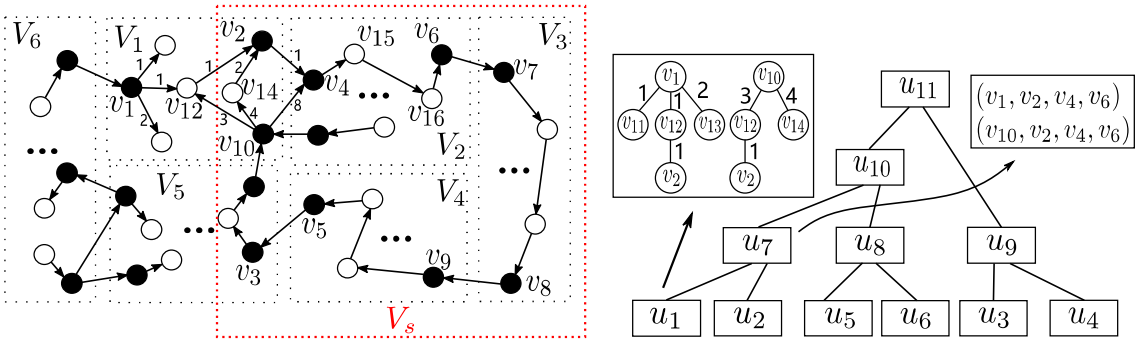
In this paper, we propose a novel index, named pb-tree, to make the shortest path query on a vertex subset more efficient for the large graphs. The pb-tree T is a binary tree to organize various vertex subsets of V such that all the vertex subsets in the same level of T form a partition of V. The partition on the lower level is essentially a refinement of that on a higher level. By pb-tree, several vertex subsets with the highest level, which are included in V_s, can be retrieved to answer the shortest path query efficiently on V_s by concatenating the shortest paths maintained in these vertex subsets.

The main contributions of this paper are summarized below. First, we study the problem of the shortest path query on a given vertex subset and develop a novel index pb-tree to solve it. We introduce how to construct pb-tree efficiently. Second, we propose an efficient query algorithm based on pb-tree to answer such shortest path query. Third, we analyze the time and space complexity for pb-tree construction and query algorithm. Forth, we conduct extensive experiments on several real-life datasets to confirm the efficiency of our method.

The rest of this paper is organized as follows. Section 2 gives the problem definition. Section 3 describes what is pb-tree and how to construct it. We introduce how to answer the shortest path query on a vertex subset using pb-tree in Sect. 4 and conduct experiments using three real-life datasets in Sect. 5. Section 6 discusses the related works. Finally, we conclude this paper in Sect. 7.

2 Problem Statement

A weighted graph is a simple directed graph denoted as $G = (V, E, w)$, where V is the set of vertices and E is the set of edges in G, each edge $e \in E$ is represented by $e = (u, v)$, $u, v \in V$, e is called u's outgoing edge or v's incoming edge and v (or u) is called u (or v)'s outgoing(or incoming) neighbor. w is a function assigning a non-negative weight to every edge in G. For simplicity, we use $w(u, v)$ to denote the weight of the directed edge $(u, v) \in E$. A path p in G is a sequence of vertices $(v_1, v_2, \cdots, v_k)$, such that (v_i, v_{i+1}) is a directed edge in G for $1 \leq i \leq k-1$. The weight of path p, denoted as $w(p)$, is defined as the sum of the weight of every edge in p, i.e., $w(p) = \sum_{1 \leq i \leq k-1} w(v_i, v_{i+1})$. Our work can be easily extended to handle the undirected graphs, in which an undirected edge (u, v) is equivalent to two directed edges (u, v) and (v, u).



(a) A graph G with six clusters (b) The pb-tree of G

Fig. 1. A graph G and the pb-tree of it (Color figure online)

In this paper, we study a special kind of shortest path query restricted to a vertex subset. Given a vertex subset $V_s \subseteq V$, an induced subgraph on V_s, denoted as $G_s(V_s, E_s)$, is a subgraph of G satisfying the two following conditions: (1) $E_s \subseteq E$; (2) for any two vertices $v_i, v_j \in V_s$, if $(v_i, v_j) \in E$, then $(v_i, v_j) \in E_s$. We say a path p is in G_s if all the vertices and edges that p passing through are in G_s. Next, we give the definition of the shortest path query on a given vertex subset V_s.

Definition 1 (The shortest path query on a vertex subset). *Given a graph $G = (V, E, w)$, a vertex subset $V_s \in V$, a source vertex v_s and a destination vertex v_e, where $v_s, v_e \in V_s$ and G_s is the induced graph on V_s, the short path query on V_s is to find a path p^* with the minimum weight $w(p^*)$ among all the paths in G_s.*

Figure 1(a) illustrates an example graph G and V_s is bounded in red dot line. The shortest path from v_{10} to v_2 on G and V_s are (v_{10}, v_{12}, v_2) and (v_{10}, v_{14}, v_2) respectively because v_{12} is not in V_s.

3 Partition-Based Tree for Shortest Path Query on a Vertex Subset

In this section, we propose a novel index, named ***Partition-Based Tree*** (or pb-tree for simplicity), to improve the efficiency of the shortest path query on a given vertex subset. A pb-tree, denoted as T, essentially is an index to organize several vertex subsets in a binary tree shape. Specifically, every leaf node in pb-tree T represents a vertex subset, and it can be regarded as a cluster of V. Thus the set of leaf nodes is a partition of V. Every non-leaf node is the super set of its two children. By pb-tree, the nodes in pb-tree which are included in a given V_s can be anchored rapidly and then they can be utilized to answer the shortest path query on V_s. In the following, we first introduce what is pb-tree and then discuss how to construct it. Finally, we explain how to partition a graph into several clusters.

Table 1. Frequently used notations

Notation	Description
$l(u_i)$	The level of the node $u_i \in T$
P_x	A shortest path tree rooted at v_x on G
S_i	The set of all the shortest path trees rooted at all the entries of u_i on G_i
$p^*_{x,y}$	The shortest path from v_x to v_y in G
$a_{x,y}$	The abstract path from v_x to v_y in G
A_i	The set of all the abstract paths for all the pairs of entry and exit in u_i

3.1 What Is Partition-Based Tree?

Definition 2 (Partition). *Given a graph $G(V, E)$, a* ***partition*** *$\mathcal{P}$ of G is a collection of k vertex subsets $\{V_1, \cdots, V_k\}$ of V, such that: (1) for $\forall V_i, V_j$ $(i \neq j)$, $V_i \cap V_j = \emptyset$; (2) $V = \bigcup_{1 \leq i \leq k} V_i$. Each $V_i \subseteq V$ is called a cluster in G. A vertex v_x is called an* entry *of cluster V_i under partition $\mathcal{P}$, if (1) $v_x \in V_i$; and (2) $\exists v_y$, $v_y \notin V_i \wedge v_y \in N^-(v_x)$. Similarly, A vertex v_x is called an* exit *of cluster V_i, if (1) $v_x \in V_i$; and (2) $\exists v_y$, $v_y \notin V_i \wedge v_y \in N^+(v_x)$. $N^-(v_x)$ and $N^+(v_x)$ are v_x's incoming and outgoing neighbor set, respectively. Entries and exits are also called the* border vertices*.*

We use *V.entry* and *V.exit* to denote the entry set and exit set of G respectively, and use V_i*.entry* and V_i*.exit* to denote the entry set and exit set of cluster V_i respectively. Obviously, $V.entry = \bigcup_{1 \leq i \leq k} V_i.entry$ and $V.exit = \bigcup_{1 \leq i \leq k} V_i.exit$.

The pb-tree T is essentially an index to organize various vertex subsets in a similar shape as a binary tree. Given a partition $\mathcal{P}$ of G, a pb-tree can be constructed. Specifically, every leaf node $u_i \in T$ corresponds to a cluster V_i under $\mathcal{P}$ and all leaf nodes consist of the partition $\mathcal{P}$. Every non-leaf node corresponds to the union of the vertex subsets represented by its two children, respectively. Each node in pb-tree has a level to indicate the location of it in the pb-tree. We use $l(u_i)$ to denote the level of the node $u_i \in T$. For every leaf node u_i in T, we set $l(u_i) = 1$. For the root node u_{root} of T, we set $l(u_{\text{root}}) = h$. Note that all the non-leaf nodes on the same level consist of a partition of G and each node can be regarded as a cluster under this partition. The partition comprised of the nodes on the low level is a refinement of the partition on the high level.

There are two kinds of information should be maintained with a pb-tree T. A ***shortest path tree set*** is maintained for every leaf node and an ***abstract path set*** is maintained for every non-leaf node. We first introduce the shortest path tree set below.

Given a connected graph $G(V, E)$ and a vertex $v_x \in V$, a shortest path tree rooted at v_x on G, denoted as P_x, is a tree such that the distance from v_x to any other vertex v_y in the tree is exactly the shortest distance from v_x to v_y in G. Every leaf node $u_i \in T$ is essentially a vertex subset of V. Let G_i denote the induced subgraph of G on u_i. The shortest path tree set S_i of u_i is

the set of all the shortest path trees rooted at all the entries of u_i on G_i, i.e., $S_i = \{P_x | v_x \in u_i.entry, P_x \subseteq G_i\}$.

We next give the definition of the abstract path for every non-leaf node in pb-tree.

Definition 3 (Abstract Path). *Given a non-leaf node $u_i \in T$, v_x and v_y are the entry and exit of u_i respectively. An abstract path from v_x to v_y, denoted as $a_{x,y}$, is a vertex sub-sequence of the shortest path $p^*_{x,y}$ from v_x to v_y in G such that all the vertices in $a_{x,y}$ are the border vertices of u_i's children.*

Based on above definition, an abstract path $a_{x,y}$ can be considered as an "abstract" of the shortest path $p^*_{x,y}$ by consisting of the border vertices of u_i's children. For every non-leaf node $u_i \in T$, its abstract path set A_i is the set of all the abstract paths for all the pairs of entry and exit in u_i, i.e., $A_i = \{a_{x,y} | v_x \in u_i.entry, v_y \in u_i.exit\}$.

Figure 1(b) shows the pb-tree of graph G in Fig. 1(a). For a leaf node u_1, a shortest path tree set is maintained for it. For a non-leaf node u_7, an abstract path set is maintained for it. For the readers convenience, Table 1 lists some frequently used notations.

3.2 How to Construct Partition-Based Tree?

As shown in Algorithm 1, the pb-tree is constructed in a bottom-up manner. Given a partition $\mathcal{P}$ of G, Algorithm 1 first calls Leaf-Node (V_i) to construct the leaf node u_i for every cluster $V_i \in \mathcal{P}$ (line 2–4). U is a temporary set to maintain all the nodes on the same level h. In each iteration, Algorithm 1 calls Non-Leaf-Node (U) to construct the non-leaf nodes on the level $h+1$ by merging the nodes on the level h (line 5–7). When U is empty, Algorithm 1 terminates and returns the pb-tree T. In the following, we introduce how to construct the leaf nodes and the non-leaf nodes by Leaf-Node (V_i) and Non-Leaf-Node (U) respectively.

Algorithm 1: Partition-Based-Tree $(G, \mathcal{P})$

Input: G, a partition $\mathcal{P}$ of G
Output: the pb-tree T based on $\mathcal{P}$.

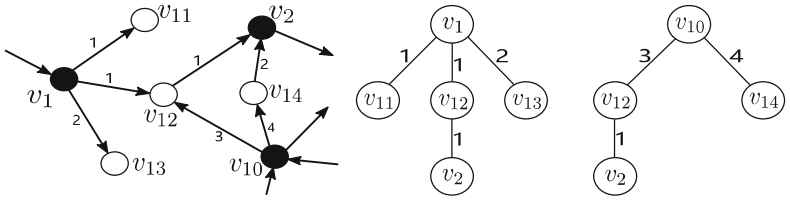
1: $T \leftarrow \emptyset, U \leftarrow \emptyset, h \leftarrow 1$;
2: **for** each cluster $V_i \in \mathcal{P}$ **do**
3: Leaf-Node (V_i);
4: $U \leftarrow U \cup \{u_i\}$;
5: **while** $U \neq \emptyset$ **do**
6: Non-Leaf-Node (U);
7: $h \leftarrow h+1, U \leftarrow U \cup T_h$;
8: **return** T

Algorithm 2: LEAF-NODE (V_i)

1: $u_i \leftarrow V_i$, $S_i \leftarrow \emptyset$;
2: **for** each $v_x \in V_i.entry$ **do**
3: computes the shortest path tree P_x rooted at v_x on G_i;
4: $S_i \leftarrow S_i \cup \{P_x\}$
5: interts u_i with S_i into T as a leaf node;

Leaf Node Construction: Given a partition $\mathcal{P}$ of G, all the clusters in $\mathcal{P}$ are the leaf nodes of T. The pseudo-code of LEAF-NODE is shown in Algorithm 2. For each cluster $V_i \in \mathcal{P}$, Algorithm 2 first sets V_i as a leaf node u_i and calculates the shortest path tree set S_i (line 1). There are several methods and we use Dijkstra algorithm to compute the shortest path tree P_x for each entry $v_x \in u_i.entry$ (line 3). Finally, Algorithm 2 inserts u_i with S_i and the crossing paths into pb-tree T as a leaf node (line 5).

Figure 2 depicts a cluster V_1 (Fig. 2(a)) and its shortest path trees rooted at two entries in V_1 (Fig. 2(b)). For example, the shortest path from v_1 to v_2 in G is exactly the simple path from v_1 to v_2 in the shortest path tree P_1.



(a) Cluster V_1 of G (b) The shortest path trees in u_1

Fig. 2. Leaf node construction

Non-leaf Node Construction: The non-leaf nodes in pb-tree T are constructed level by level. A temporary set U is utilized to maintain all the nodes on the level h which have been constructed in T and then Algorithm 3 constructs all the non-leaf nodes on the level $h+1$ by merging two nodes with the maximum size of ***crossing edge set*** in U iteratively. A crossing edge set between node u_i and u_j on the same level of T, denoted as $C_{i,j}$, is the set of all the crossing edges between u_i and u_j, i.e., $C_{i,j} = \{(v_x, v_y) | v_x \in u_i \wedge v_y \in u_j \text{ or } v_x \in u_j \wedge v_y \in u_i\}$. It is worth noting that $C_{i,j} = C_{j,i}$. In each iteration, two nodes u_i and u_j with the maximum $|C_i, j|$ in U are merged into a new node u_k. Note that the $u_k.entry$ and $u_k.exit$ are the subset of $u_i.entry \cup u_j.entry$ and $u_i.exit \cup u_j.exit$ respectively. It is because some of the entries and exits of u_i (or u_j) become the internal vertices of u_k after merging u_i and u_j. Algorithm 3 computes the abstract path set A_k for u_k by Dijkstra algorithm. Finally, u_k is inserted into T with A_k as the parent of u_i and u_j. Note that there may be only one node u_i in U in the final iteration. In this case, u_i will be left in U and be used for constructing the level $h+2$ of T with all the non-leaf nodes on the level $h+1$.

Algorithm 3: NON-LEAF-NODE (U)

1: **while** $|U| > 1$ **do**
2: selects u_i and u_j with the maximum $|C_{i,j}|$ from U;
3: $u_k \leftarrow u_i \cup u_j$;
4: computes the abstract path set A_k for u_k;
5: inserts u_k with A_k into T as the parent node of u_i and u_j;
6: $U \leftarrow U \setminus \{u_i, u_j\}$

Figure 3 shows the construction of the pb-tree. The leaf nodes are constructed in the same way as Fig. 2. U is $\{u_1, u_2, u_3, u_4, u_5, u_6\}$ in the begining. The algorithm first merges u_1 and u_2 into u_7, because $|C_{1,2}| = 3$ is maximum. The entries of u_7 is v_1 and v_{10}. The algorithm uses the Dijkstra algorithm to compute the shortest paths from v_1 and v_{10}. The shortest path from v_1 to v_6 on V_7 is $(v_1, v_{12}, v_2, v_4, v_{15}, v_{16}, v_6)$, and the abstract path $a_{1,6}$ is (v_1, v_2, v_4, v_6). In the same way, $a_{10,6}$ is maintained as (v_{10}, v_2, v_4, v_6). After that, u_8 is mergred by u_5 and u_6. u_9 is merged by u_3 and u_4. To construct the T_3, U is $\{u_7, u_8, u_9\}$. The algorithm merges u_7 and u_8 into u_{10}, cause $|C_{7,8}| = 2$ is larger than $|C_{8,9}|$ and $|C_{7,9}|$. After that, U is $\{u_9\}$, and u_9 is used to construct the higher level of T. Then U is $\{u_9, u_{10}\}$. u_{11} is constructed by merging u_9 and u_{10}. The abstract paths in those nodes are computed in the same way as computing the abstract paths in u_7.

3.3 How to Partition Graph to Several Clusters

There are several ways to partition a graph to several clusters. For different partitions, the number of entries and exits are different. In our problem, the fewer number of entries and exits makes the smaller size of pb-tree index. Intuitively, the fewer edges among different clusters result in the less number of entries and exits in graphs. Thus it is a problem to find an optimal partition such that the edges among different clusters are sparse and the edges in the same cluster are dense. This problem has been well studied, and there are many effective and efficient algorithms[1,4,17] to solve it. In this paper, We adopt the METIS algorithm[1], which is a classic graph partition algorithm.

3.4 Complexity Analysis

For graph G, let m be the number of edges, k be the number of clusters, α and β be the maximum number of vertices and edges inside the cluster and a be the maximum number of the borders in each cluster.

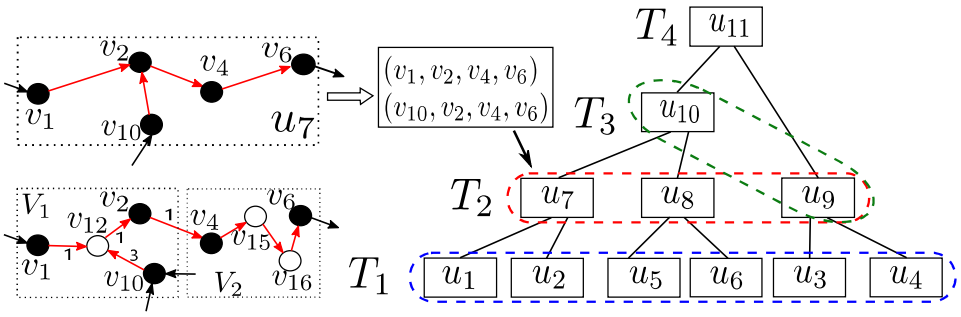


Fig. 3. pb-tree construction

Time Complexity: For each leaf node, the shortest path tree set can be built in $O(a(\alpha \log \alpha + \beta))$ time. All the leaf nodes can be constructed in $O(k\alpha^2 log\alpha + k\alpha\beta)$. As a binary tree, the maximum level of pb-tree is $logk + 1$ and the maximum number of the non-leaf nodes on level h is $\frac{k}{2^{h-1}}$. A non-leaf node on level h is constructed by merging two children. The entry and the exit sets of the two children can be merged in $O(1)$. The time complexity of searching all the neighbors of borders on level h is $O(m)$. The number of borders in a non-leaf node on level h is $O(2^{h-1}a)$, and the number of the vertices in it is $O(2^{h-1}a)$ because abstract paths are computed only by the borders of the children. The Dijkstra algorithm is utilized to compute the abstract paths from each entry. Computing all the abstract paths in a non-leaf node on level h is in $O(2^{h-1}a(2^{h-1}alog(2^{h-1}a) + 2^{2h-2}a^2)) = O(8^{h-1}a^3)$. The time complexity of constructing the non-leaf nodes on level h is $O(4^{h-1}ka^3 + m)$. Because there are $\log k + 1$ levels in pb-tree, then we have

$$\sum_{h=2}^{logk+1} (4^{h-1}ka^3 + m) = mlogk + \frac{4}{3}ka^3(4^{logk} - 1) = mlogk + \frac{4}{3}ka^3(k^2 - 1)$$

Thus the time complexity of constructing all the non-leaf nodes and pb-tree are $O(mlogk + k^3a^3)$ and $O(k\alpha\beta + mlogk + \alpha^3k^3)$ respectively.

Space Complexity: In the worst-case, each shortest path tree in a leaf node contains all the vertices and edges in that node. The number of the shortest path trees in a leaf node is $O(a)$; thus the number of vertices in each leaf node is $O(a\alpha) = O(\alpha^2)$ and the number of edges is $O(a\beta) = O(\alpha\beta)$. The space complexity of leaf nodes is $O(k\alpha^2 + k\alpha\beta)$. For a non-leaf node on level h, the number of the borders is $2^{h-1}a$, and the number of the abstract paths is $O(4^{h-1}a^2)$. In the worst-case, each abstract path contains all the vertices in the node. The space complexity of the non-leaf nodes on level h is $O(a^3 4^{h-1}k)$. Because there are $\log k + 1$ levels in pb-tree, then we have

$$\sum_{h=2}^{logk+1} (a^3 4^{h-1}k) = \frac{4}{3}ka^3(4^{logk} - 1) = \frac{4}{3}ka^3(k^2 - 1)$$

Algorithm 4: QUERY-PROCESSING $(q = (V_s, v_s, v_e))$

Input: V_s, v_s, v_e, pb-tree T, G
Output: $p^*_{s,e}$

1: $Q \leftarrow V_s, \tau_s \leftarrow 0$
2: **while** $v_e \in Q$ **do**
3: gets v_x from Q with minimum τ_x
4: **if** v_x is expanded by $a_{y,x}$ **then**
5: PATH-RECOVER $(Q, a_{y,x})$
6: **if** $v_x \in u_i.entry$ **then**
7: **if** u_i is a complete node **then**
8: NODE-IDENTIFY (Q, v_x, u_i)
9: **else**
10: PARTIAL-SEARCH (Q, v_x, u_i)
11: **if** $v_x \notin u_i.entry \vee v_x \in u_i.exit$ **then**
12: updates the τ of v_x's outgoing neighbors in Q;
13: dequeues v_x from Q
14: **return** $p^*_{s,e}$

The space complexity of constructing all the non-leaf nodes is $O(k^3a^3)$, and the space complexity of constructing the pb-tree is $O(k^3\alpha^3 + k\alpha\beta)$.

4 Query Processing by pb-tree

4.1 Querying Algorithm

In this section, we introduce how to find the shortest path on a given vertex subset by pb-tree. For a vertex subset V_s, all the nodes in pb-tree can be divided into three categories: ***Complete node***, ***Partial node*** and ***Irrelevant node***. A node $u \in T$ is a complete node for V_s if all the vertices in u are included in V_s, and it is a partial node if there exists a proper vertex subset of u included in V_s. Correspondingly, u is an irrelevant node if all the vertices in u are outside of V_s. Note that if a node is a complete node, then all its descendant are complete nodes. We propose a Dijkstra-based algorithm on pb-tree to make the query more efficient by expanding the abstract paths in complete non-leaf nodes and the shortest path trees in partial leaf nodes.

The querying algorithm is shown in Algorithm 4. Algorithm 4 utilizes a prior queue Q to iteratively dequeue the vertices in V_s until the ending vertex v_e is dequeued. In each iteration, a vertex v_x is dequeued from Q with the minimum τ_x, where τ_x is the distance from the starting vertex v_s to it. Initially, Q is set as V_s. τ_s is 0 and τ_x is ∞ for other vertices in V_s. If v_x is an exit and it is expanded by an abstract path $a_{y,x}$, Algorithm 4 calls PATH-RECOVER to dequeue all the vertices in the path represented by $a_{y,x}$ from Q (line 4–5) and then updates τ_z for every v_x's outgoing neighbor v_z in Q (line 12). If v_x is an entry of a leaf node u_i, Algorithm 4 calls NODE-IDENTIFY to find the complete node u_j with the highest level such that v_x is still an entry of u_j. NODE-IDENTIFY uses the

Algorithm 5: PARTIAL-SEARCH (Q, v_x, u_i)

```
1: for each v_y ∈ P_x do
2:    if v_y ∉ V_s then
3:       deletes the branch below v_y;
4:    else
5:       if v_y ∈ Q then
6:          updates p*_{s,y} and τ_y;
```

Algorithm 6: PATH-RECOVER $(Q, a_{y,x})$

```
1: for each a_{i,i+1} ⊂ a_{y,x} do
2:    if ∃ a child node u_k of u_j, a_{i,i+1} ∈ A_k then
3:       PATH-RECOVER (a_{i,i+1});
4:    else
5:       gets p*_{i,i+1} by searching P_i
6:       Q ← Q \ p*_{i,i+1}
```

abstract paths to expand the exits of u_j and then updates Q (line 8). Note that u_j may be the leaf node u_i. If such u_j does no exist, the leaf node u_i is a partial node, then Algorithm 4 calls PARTIAL-SEARCH to expand the vertices in u_i and updates Q (line 10). If v_x is not an entry or exit, it must be an internal vertex in a leaf node u_i, then Algorithm 4 updates v_x's outgoing neighbors in Q in the similar way as Dijkstra algorithm (line 12). Algorithm 4 terminates when the ending vertex v_e first dequeued from Q and the τ_e is the shortest distance from v_s to v_e on V_s. Next, we introduce NODE-IDENTIFY, PARTIAL-SEARCH and PATH-RECOVER respectively.

Partial Search: For a partial node u_i and an entry v_x, the shortest path tree P_x of S_i is utilized to expand the shortest paths. Algorithm 5 utilizes BFS to search P_x. For every $v_y \in P_x$, if $v_y \notin V_s$, the branch below v_y can be ignored (line 3). And if $v_y \in Q$, the Algorithm 5 updates the $p^*_{s,y}$ and τ_y (line 6).

Path Recover: For an abstract path $a_{y,x}$ of a complete node u_j, PATH-RECOVER computes $p^*_{y,x}$ in the descendant nodes of u_j. As shown in Algorithm 6, for each sub-abstract path $a_{i,i+1}$ of $a_{y,x}$ which can be found in one of u_j's children, Algorithm 6 calls PATH-RECOVER in that child to compute the $a_{i,i+1}$ (line 3). Otherwise, Algorithm 6 searches the shortest path tree P_i to compute $p^*_{i,i+1}$ (line 5).

Node Identify: Given a leaf node u_i and a vertex $v_x \in u_i.entry$, the pseudocode of NODE-IDENTIFY is shown in Algorithm 7. It first finds the parent node of u_i. If v_x is an entry of the parent node, and it is a complete node, Algorithm 7 checks the parent node of it in the same way (line 3–4). u_j is the complete node on the highest level and v_x is an entry of it. Then Algorithm 7 searches the shortest path tree P_x to update the exits of u_j in Q, if u_j is a leaf node (line 7). If u_j is a non-leaf node, Algorithm 7 utilizes the abstract paths which start from v_x to update the exits of u_j in Q (line 9).

Algorithm 7: NODE-IDENTIFY (v_x, u_i)

finds the parent node u_j of u_i;
while $v_x \in u_j.entry$ and u_j is a complete node **do**
    $u_i \leftarrow u_j$
    finds the parent node u_j of u_i;
$u_j \leftarrow u_i$
if u_j is a leaf node **then**
    updates the τ of the exits in Q using the shortest path trees in u_j;
else
    updates the τ of the exits in Q using the abstract paths in u_j;

4.2 Example

Figure 4 shows the query process from v_{10} to v_3 on V_s. Initially, Q is set as V_s and τ_{10} is set to 0. In the 1st iteration, v_{10} is dequeued from Q. v_{10} is an entry of a partial node u_1. The algorithm searches the shortest path tree P_{10} in u_1. Because v_{12} is not in V_s, only τ_{14} is updated to 4 and $p^*_{10,14}$ is updated to $(10, 14)$. v_{10} is also an exit of u_1, τ_4 is updated to 8 and $p^*_{10,4}$ is updated to $(10, 4)$. In the 2nd iteration, v_{14} is dequeued from Q. Cause v_{14} is not an entry of u_1, algorithm updates the $p^*_{10,2}$ to $(10, 14, 2)$ by the edge (v_{14}, v_2), and τ_2 is updated to 6. As an exit of u_1, the algorithm searches the outgoing neighbors of v_2 in the 3rd iteration. τ_4 is updated to 7 and $p^*_{10,4}$ is updated to $(10, 14, 2, 4)$. This is the (1) of Fig. 4. In the 4th iteration, because v_4 is an entry of u_2, and it is not an entry of u_7, P_4 in u_2 is searched and τ_6 is updated. In the same way, τ_7 is updated in the next iteration, and $p^*_{10,7}$ is updated to $(10, 14, 2, 4, 15, 16, 6, 7)$. Then v_7 is dequeued from Q. It is an entry of a complete node u_3. The algorithm searches the pb-tree and finds the complete node u_9 with the highest level such that v_7 is an entry of it. That is the (2) of Fig. 4. In u_9, the abstract path $a_{7,5}$ is utilized to update the $p^*_{10,5}$ and τ_5. In the next iteration, v_5 is dequeued from Q and it is expanded by $a_{7,5}$. The algorithm computes $p^*_{7,5}$ by computing $p^*_{7,8}$ in u_3 and $p^*_{9,5}$ in u_4. u_3 and u_4 are the leaf nodes. Therefore, the algorithm searches P_7 in u_3 and P_9 in u_4. All the vertices in $p^*_{7,5}$ are dequeued from Q. After that, τ_3

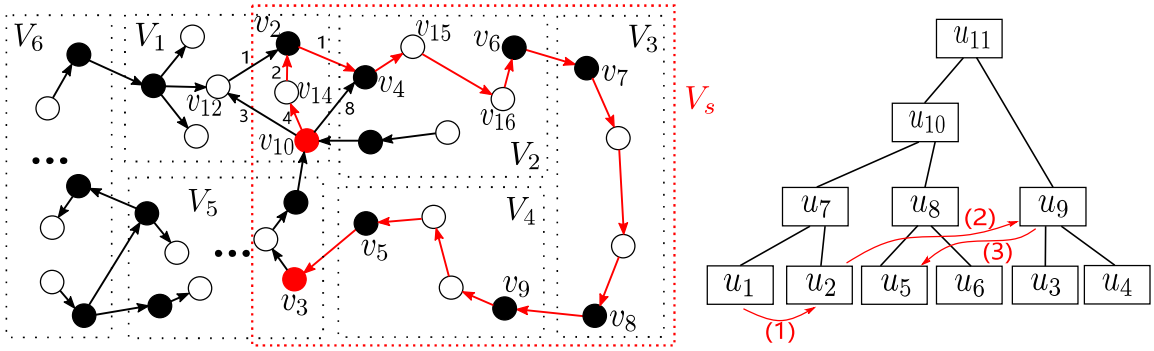


Fig. 4. A shortest path query from v_{10} to v_3 on V_s

and $p^*_{10,3}$ are updated by the edge (v_5, v_3). This is the (3) of Fig. 4. In the next iteration, v_3 is dequeued from Q. The algorithm terminates and returns $p^*_{10,3}$.

4.3 Complexity Analysis

Time Complexity: For a graph G and a vertex subset V_s, let r and q be the number of vertices and edges in G_s, k be the number of clusters, α and a be the maximum number of vertices and borders in each cluster, ψ_c and ψ_p be the number of the complete leaf nodes and the partial leaf nodes. The identification of each node is in $O(1)$, and the number of nodes in pb-tree is $2k-1$. The time complexity of identifying all the nodes is $O(2k-1) = O(k)$. For an entry of a complete leaf node, the algorithm finds the complete nodes on a higher level in $O(logk+1) = O(logk)$, when all the complete nodes are on the highest level. For a complete node on level $logk+1$, the number of the borders is $O(ka)$. Searching the abstract paths from that vertex is in $O(ka)$. Thus the time complexity of expanding the shortest paths from an entry of a complete leaf node is $O(logk + ka)$. For an entry of a partial leaf node, the algorithm searches the shortest path tree in that node in $O(\alpha)$. For the exits of the complete leaf nodes and the vertices in partial leaf nodes, whose number is $O(\psi_c a + \psi_p \alpha)$, the algorithm searches all the neighbors of it in V_s in $O(\psi_c ar + \psi_p \alpha r)$. For an abstract path, in the worst-case, it is in the node on level $logk+1$, and all the borders of that node are in the path. The number of the borders is $O(ka)$, and for each pair of the borders, the Path-Recover searches a shortest path tree in a leaf node. The time complexity of computing such an abstract path is $O(ka\alpha)$. The number of the borders in complete nodes is $\psi_c a$. To sum up, because $\psi_c + \psi_p = k, \alpha \geq a$, the time complexity of our method is $O(\psi_c a(logk + ka) + \psi_p a\alpha + \psi_c ar + \psi_p \alpha r + \psi_c a\alpha ka + k) = O(k\alpha r + k^2\alpha^3)$. In the worst-case, the number of the complete nodes is zero, and all the shortest path trees can not be utilized. Then time complexity is $O(\psi_p \alpha r) = O(rlogr + q)$, which is the time complexity of the Dijkstra algorithm. In practice, the complexity is much smaller than the worst-case complexity.

Space Complexity: The query algorithm maintains a prior queue Q. In the worst-case, all the vertices in V_s will be dequeued from Q. Therefore, the space complexity is the number of vertices in V_s, i.e. $O(r)$.

5 Experiements

In this section, we study the performance of our algorithm on three real-life network datasets. Section 5.1 explains the datasets and experimental settings. Section 5.2 presents the performance of the algorithms.

5.1 Datasets and Experimental Settings

Experimental Settings. All the experiments were done on a 2.50 GHz Intel(R) Xeon(R) Platinum 8255C CPU with 128G main memory, running on Linux VM-16-3-ubuntu 4.4.0-130-generic. All algorithms are implemented by C++.

Datasets. We use three real road networks from the 9th DIMACS Implementation Challenge (http://users.diag.uniroma1.it/challenge9/download.shtml). All the edges of them are the real roads in the corresponding areas. Table 2 summarizes the properties of the datasets. $|V|$ and $|E|$ are the number of vertices and edges in the road network.

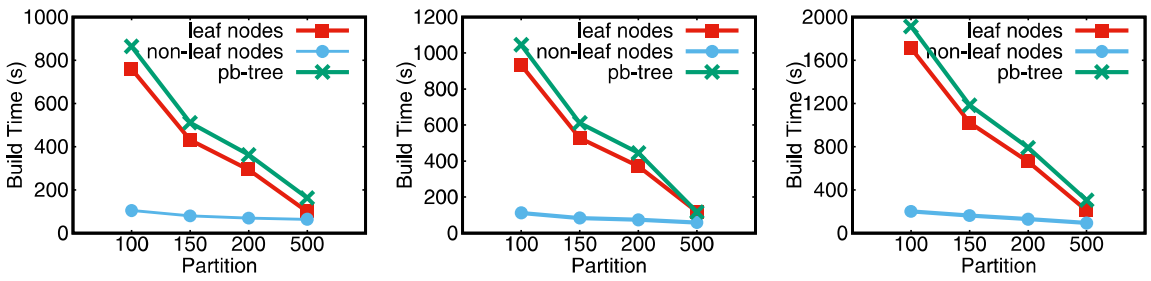
Table 2. Datasets

Dataset	$\|V\|$	$\|E\|$	Description
NY	264,346	733,846	New York City Road Network
BAY	327,270	800,172	San Francisco Bay Area Road Network
COL	435,666	1,057,066	Colorado Road Network

Query Set. For each dataset, we use four different kinds of partitions, which partition the graph into 100, 150, 200, and 500 clusters respectively. We construct a pb-tree for each partition. We study the query performance by varying vertex subset V_s. We test 9 kinds of queries, where every query set is a set of queries with a same size of vertex subsets. These vertex subsets contain the 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% and 90% vertices randomly taken from V. For each query set, we test 100 random queries and report the average querying time as the results for the current query set.

5.2 Experimental Results

Exp-1. Build Time of Index. Figure 5 shows the build time of the pb-tree based on different number of clusters. Observe that, as the number of the clusters increases, the build time of the leaf nodes and the non-leaf nodes are decreased. The main reasons are as follow. As the number of leaf nodes increases, the number of the vertices in a single leaf node decreases. And the time to build the shortest paths tree also decreases. For non-leaf nodes, although the number of all the non-leaf nodes increases, the time to compute the abstract paths based on the Dijkstra algorithm also decreases.



(a) NY (b) BAY (c) COL

Fig. 5. Build time of pb-tree

Exp-2. Index Size. Figure 6 shows the size of the pb-tree based on different number of clusters. As the number of the clusters increases, though the size and the number of non-leaf nodes are increased, the size of leaf nodes and the pb-tree are decreased. The main reasons are as follow. On the one hand, as the number of the vertices in a leaf node decreases, the number of the vertices maintained in the shortest path trees also decreases. On the other hand, for more clusters, there will be more non-leaf nodes and more abstract paths in them. Therefore, the size of the non-leaf nodes increases.

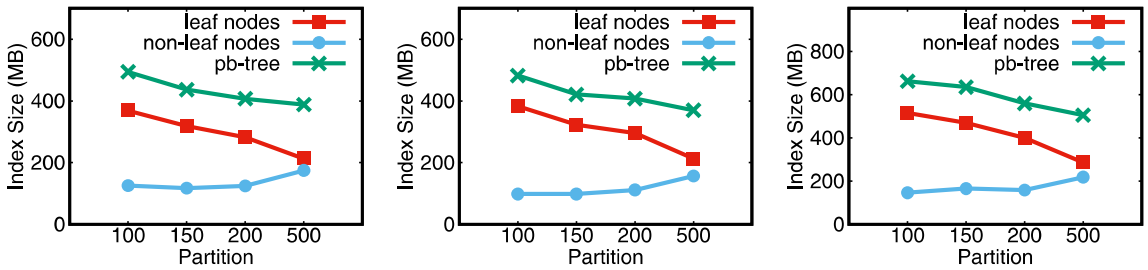


Fig. 6. Index size of pb-tree

Exp-3. Query Time Based on Different Number of Clusters. Figure 7 shows the query time using the four pb-trees based on 100, 150, 200 and 500 clusters on three datasets in four kinds of vertex subsets whose number of the vertices are 90%, 80%, 70% and 60% of the number of the vertices in V. We made two observations. The first one is that for the vertex subsets with the same size, the more clusters the pb-tree is based on, the less time the query processing will take. The reason is that there are more complete nodes and abstract paths can be used. Secondly, as the size of the vertex subsets decreases, the query time also decreases. This is mainly because as the size of the vertex subsets decreases, more vertices are unreachable, and the query results can be returned faster.

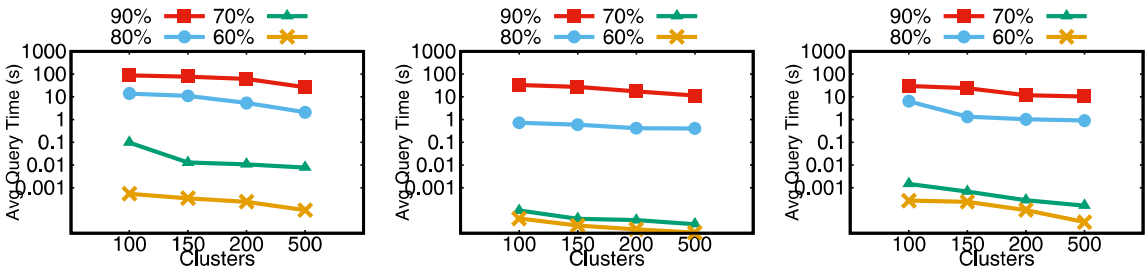


Fig. 7. Query time based on different number of clusters

Exp-4. Query Efficiency. We compared our algorithm with Dijkstra algorithm. Table 3 shows the query times using the pb-tree based on 100 clusters and

the Dijkstra algorithm. On each dataset, we find that the Dijkstra algorithm takes more time than our method on each size of vertex subset. We also find that when the size of the vertex subset is 80% of the vertices in V, the time difference between the two methods is minimal. That is because most of the nodes in pb-tree are partial nodes in that case, so most of the abstract paths can not be used. When the size of the vertex subset drops below 50% of the vertices in V, the query times for both methods tend to stabilize. That mainly because most of the vertices are unreachable, and the results can be returned quickly. For our method, the shortest path trees in each leaf node can be used, which makes our method have a better performance than the Dijkstra algorithm.

Table 3. Query times(s)

Dataset	Method	90%	80%	70%	60%	50%	40%	30%	20%	10%
NY	pb-tree	87.0191	13.8673	9.65E-02	5.44E-04	6.48E-05	1.65E-05	2.59E-05	8.78E-05	2.08E-05
	Dijkstra	969.062	102.884	10.26720	4.11E-02	5.74E-04	4.94E-04	2.67E-04	1.60E-04	2.40E-04
BAY	pb-tree	32.8067	0.72000	9.86E-05	4.38E-05	3.22E-05	8.12E-05	3.59E-05	5.77E-05	2.02E-05
	Dijkstra	542.614	10.1591	4.34E-02	9.45E-04	4.52E-04	5.02E-04	1.74E-04	2.91E-04	3.00E-04
COL	pb-tree	30.3478	6.36190	1.48E-03	2.69E-04	6.45E-05	4.98E-05	8.59E-05	5.77E-05	4.85E-05
	Dijkstra	743.509	90.7764	0.494680	8.02E-03	1.15E-03	1.87E-03	1.76E-03	1.21E-03	9.47E-04

6 Related Work

In this section, we will mainly discuss two categories of related work: the first one is the existing algorithms for answering unconstrained shortest path queries; the other one is existing approaches for answering constrained shortest path queries.

In the first category, the traditional shortest path query algorithms, such as the Dijkstra algorithm[5], can be used to solve the problems we supposed in this paper. But it will take a long time to answer the query. The shortest path quad tree is proposed in [13]. Xiao et al. in [16] proposes the concept of the compact BFS-trees. A novel index called TEDI has been proposed by Wei et al. in [15]. It utilizes the tree decomposition theory to build the tree. Ruicheng Zhong et al. propose a G-Tree model in [18]. Goldberg et al. in [6] propose a method which is to choose some vertices as landmark vertices and store the shortest paths between each pair of them. Qiao et al. in [11] propose a query-dependent local landmark scheme. [2] proposes another novel exact method based on distance-aware 2-hop cover for the distance queries. However, those methods can not be used to answer the problem we proposed in this work because the vertices in those pre-computed paths maybe not in the given vertex subset.

In the second category, several works [7,9,10,14] study the constrained shortest path problem on a large graph. A Lagrangian relaxation algorithm for the problem of finding a shortest path between two vertices in a network has been developed in [7]. H. C. Joksch et al. propose a linear programming approach and a dynamic programming approach in [9]. Kurt Mehlhorn et al. present the

hull approach, a combinatorial algorithm for solving a linear programming relaxation in [10]. [14] tackles a generalization of the weight constrained shortest path problem in a directed network. Some works aim to answer the query with the constrained edges. Bonchi et al. propose an approximate algorithm for shortest path query with edge constraints in [3]. But the algorithm can not support the exact shortest path query. Michael N. Rice et al. propose a method by precalculating the paths between some of the two vertices with the label of the edge in [12]. Mohamed S. Hassan et al. construct an index called EDP, which is one of the state-of-art methods to answer the query with the constrained edges in [8]. The EDP contains many subgraphs with the same label of edges and stores every shortest path from an inner vertex to a border vertex in each subgraph. Those methods can not be used to solve the problem we proposed in this paper, cause the vertex subset V_s is not constrained by labels or another weight of the edges.

7 Conclusion

In this paper, we study the problem of the shortest path query on a vertex subset. We first give the definition of the shortest path query on a vertex subset. Second, we propose a novel index named pb-tree to facilitate the shortest path query on a vertex subset. We introduce what the pb-tree is and how to construct it. We also introduce how to utilize our index to answer the queries. Finally, we confirm the effectiveness and efficiency of our method through extensive experiments on real-life datasets.

Acknowledgments. This work is supported by the National Key Research and Development Project 2019YFB2101903 and the National Natural Science Foundation of China No. 61402323, 61972275.

References

1. Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006) Proceedings, Rhodes Island, Greece, 25–29 April 2006. IEEE (2006)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 349–360 (2013)
3. Bonchi, F., Gionis, A., Gullo, F., Ukkonen, A.: Distance oracles in edge-labeled graphs. In: Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, 24–28 March 2014, pp. 547–558 (2014)
4. Dhillon, I.S., Guan, Y., Kulis, B.: Weighted graph cuts without eigenvectors a multilevel approach. IEEE Trans. Pattern Anal. Mach. Intell. **29**(11), 1944–1957 (2007)
5. Dijkstra, E.W., et al.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)

6. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a search meets graph theory. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, 23–25 January 2005, pp. 156–165 (2005)
7. Handler, G.Y., Zang, I.: A dual algorithm for the constrained shortest path problem. Networks **10**(4), 293–309 (1980)
8. Hassan, M.S., Aref, W.G., Aly, A.M.: Graph indexing for shortest-path finding over dynamic sub-graphs. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1183–1197 (2016)
9. Joksch, H.C.: The shortest route problem with constraints. J. Math. Anal. Appl. **14**(2), 191–197 (1966)
10. Mehlhorn, K., Ziegelmann, M.: Resource constrained shortest paths. In: Paterson, M.S. (ed.) ESA 2000. LNCS, vol. 1879, pp. 326–337. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45253-2_30
11. Qiao, M., Cheng, H., Chang, L., Yu, J.X.: Approximate shortest distance computing: a query-dependent local landmark scheme. In: IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1–5 April 2012, pp. 462–473 (2012)
12. Rice, M.N., Tsotras, V.J.: Graph indexing of road networks for shortest path queries with label restrictions. PVLDB **4**(2), 69–80 (2010)
13. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, 10–12 June 2008, pp. 43–54 (2008)
14. Smith, O.J., Boland, N., Waterer, H.: Solving shortest path problems with a weight constraint and replenishment arcs. Comput. OR **39**(5), 964–984 (2012)
15. Wei, F.: TEDI: efficient shortest path query answering on graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp. 99–110 (2010)
16. Xiao, Y., Wu, W., Pei, J., Wang, W., He, Z.: Efficiently indexing shortest paths by exploiting symmetry in graphs. In: 12th International Conference on Extending Database Technology, EDBT 2009, Saint Petersburg, Russia, 24–26 March 2009, Proceedings, pp. 493–504 (2009)
17. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.J.: SCAN: a structural clustering algorithm for networks. In: Berkhin, P., Caruana, R., Wu, X. (eds.) Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, 12–15 August 2007, pp. 824–833. ACM (2007)
18. Zhong, R., Li, G., Tan, K., Zhou, L.: G-tree: an efficient index for KNN search on road networks. In: CIKM, pp. 39–48 (2013)