Top-k Critical Vertices Query on Shortest Path

Jing Ma, Bin Yao, *Member, IEEE*, Xiaofeng Gao, *Member, IEEE*, Yanyan Shen, and Minyi Guo, *Fellow, IEEE*

Abstract—Shortest path query is one of the most fundamental and classic problems in graph analytics, which returns the complete shortest path between any two vertices. However, in many real-life scenarios, only critical vertices on the shortest path are desirable and it is unnecessary to search for the complete path. This paper investigates the shortest path sketch by defining a top-k critical vertices (kCV) query on the shortest path. Given a source vertex s and target vertex t in a graph, kCV query can return the top-k significant vertices on the shortest path $SP(s,t)$. The significance of the vertices can be predefined. The key strategy for seeking the sketch is to apply off-line preprocessed distance oracle to accelerate on-line real-time queries. This allows us to omit unnecessary vertices and obtain the most representative sketch of the shortest path directly. We further explore a series of methods and optimizations to answer kCV query on both centralized and distributed platforms, using exact and approximate approaches, respectively. We evaluate our methods in terms of time, space complexity and approximation quality. Experiments on large-scale real-world networks validate that our algorithms are of high efficiency and accuracy.

Index Terms—kCV query, shortest path sketch, road network, social network, web graph

1 Introduction

Recent researches have concentrated on statistical characteristics of networked systems such as social networks [1], web graphs [2] and road networks [3], [4], [5], [6], [7]. Among them, point-to-point shortest distance and path queries are fundamental problems for numerous applications [8], [9], [10], [11], [12]. However, traditional solutions such as Dijkstra algorithm [13] cannot be widely applied due to its inefficiency on sizable graphs. Hence, various approaches have been proposed to efficiently solve shortest path problem in big graphs [14], [15], [16], [17]. State-of-the-art shortest path algorithms [18], [19], [20] mainly aim to find the whole path, whereas at most time, people are only interested in several crucial parts (e.g., a few key vertices) on the shortest path. For example, most travelers only care about trade centers or transportation hubs in their routes. Based on this phenomenon, researchers start to focus on the algorithms of shortest path sketch. A recent one is k-skip [21] query, which returns P^*, a subset of the vertices on the shortest path P. P^* contains at least one vertex from every k consecutive vertices in P.

This paper studies another variant of the shortest path sketch. We propose a top-k critical vertices (kCV) query on the shortest path between a given pair of vertices, which can return the top-k significant vertices on the shortest path. The significance of each vertex can be determined by several factors, and we will introduce them in detail later. kCV query can benefit people in many ways. For instance, on road networks, kCV query can help the travelers to identify the shortest path in a fast way, because in most cases the travelers already have a previous knowledge of the most important vertices on the shortest path, which are often the well-known places. Besides, sometimes when two friends need to meet each other in some landmarks, kCV query is a good way for them to find a list of popular places on their shortest path to meet, for example, a big shopping mall. On social networks, those people with lots of followers can be regarded as high-significance vertices. Users are interested in the social stars and celebrities on their chains of connection. We denote the k critical vertices returned by kCV query as kCV *objects*.

Fig. 1 illustrates an example of a 3CV query with source vertex s and target vertex t on their shortest path $SP(s,t)$. Suppose the upper vertices are more significant, the query returns critical vertices v_1, v_2, v_3 successively.

In this paper, we study on different methods to answer kCV query on centralized and distributed platforms respectively and implement them with high efficiency. Vertices should be ordered according to the definition of *significance*. Intuitively, significance describes the importance of each vertex and can be defined by various criteria. For example, the degree or betweenness centrality of vertices could be a good indicator for vertex significance.

In order to answer kCV query, we adapt label-based strategy [22][23] for the preprocessing. These are feasible labeling algorithm based on distance oracle. Distance oracle

- *J. Ma is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200000, China, the State Key Laboratory of Software Development Environment, Beihang University, Beijing 100083, China, and the Guizhou Provincial Key Laboratory of Public Big Data, Guizhou University, Guiyang, China. E-mail: heather@sjtu.edu.cn.*
- *B. Yao is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200000, China, and the Beijing Key Laboratory of Big Data Management and Analysis Methods, Renmin University of China, Beijing 100872, China. E-mail: yaobin@cs.sjtu.edu.cn.*
- *X. Gao, Y. Shen, and M. Guo are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200000, China. E-mail: {gao-xf, shen-yy, guo-my}@cs.sjtu.edu.cn.*

Manuscript received 10 Mar. 2017; revised 10 Dec. 2017; accepted 6 Feb. 2018. Date of publication 22 Feb. 2018; date of current version 10 Sept. 2018. (Corresponding author: Bin Yao.)
Recommended for acceptance by J.-R. Wen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2018.2808495

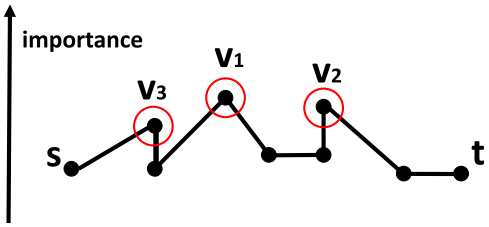


Fig. 1. 3CV query on SP(s,t).

is an auxiliary data structure generated in preprocessing in order to accelerate distance queries between any pairs of vertices. A practical distance oracle is constructed with affordable time and space cost, and it is supposed to answer the distance query with high accuracy in constant time. If taking vertex order into consideration, distance oracle could be utilized to answer kCV query by storing the highest significance vertex on the shortest path of every pair, which is the fundamental idea of our basic method to answer kCV query.

In most cases, for a distance oracle, more efficient query requires longer preprocessing time, and higher accuracy results need more time and space consumption. By different ways of constructing the distance oracle, the trade-off between accuracy and cost in time and space, and the trade-off between time consumption in preprocessing and query could be both adjusted.

The above basic method has some unnecessary overhead in the query (see details in Section 4.3.2), and hence we propose an optimization to improve the efficiency by modifying the way of distance oracle construction. With this optimization, we can save the query time by reducing some extra operations, and this acceleration of query is clearly observed in the experiments on real networks. But on the other hand, it costs higher time and space consumption in preprocessing.

To achieve better performance, we propose two parallel approaches to kCV query. The first method utilizes multi-threading technique for parallel acceleration. With a slight loss of optimality, the multi-threading method can improve the efficiency dramatically with the increment of the number of threads. Second, we leverage specialized distributed processing systems [24], [25], [26], [27], [28] for acceleration, in which delicate designs are provided to fully utilize the features of the distributed framework. In order to better utilize the capability of distributed processing, we also explore and implement top-k batch query (kBCV query) distributively, which can afford a large number of approximate kCV queries on all pairs in a set of vertices, applying reverse labeling and pruning methods.

In all, the main contributions of our work can be summarized as follows:

- We propose kCV query and explore a series of relevant techniques to solve it. This is the first study about kCV query to the best of our knowledge.
- We explore a basic method to answer kCV query and implement it on centralized platform, which computes the score of the vertex significance, sorts and outputs the kCV objects on shortest path with time complexity $O(dk|L|)$, where d stands for the average degree of vertices in the graph.
- We propose pure-labeling method, which also returns exact kCV objects and performs faster than basic method by a constant factor d, reduces time complexity of query from $O(dk|L|)$ to $O(k|L|)$, but requires longer preprocessing time.
- We design a parallel method with multi-threading technique. Compared with the above methods returning exact answers, multi-threading method is much more efficient but incurs a little sacrifice of accuracy (but also acceptable). The speedup can reach $k/\log(\alpha k)$, where α is a parameter which determines the tradeoff between time and accuracy.
- We explore kBCV query, which can support quantities of kCV queries on every pair of vertices in a set distributively. By adopting distributed computing, experimental results show that the algorithm is of good efficiency.

The rest of this paper is organized as follows. Section 2 shows some related work. Section 3 gives several definitions mentioned in this paper. Section 4 studies techniques for kCV query on centralized platform, while Section 5 introduces kBCV query on distributed platform. Section 6 presents our experimental evaluations. We make a conclusion in Section 7.

2 Related Work

2.1 Dijkstra and Bi-Directional Dijkstra

Dijkstra. Dijkstra [13] is a classical algorithm to solve shortest path problem, which can be applied to graphs with non-negative edges. Given an input directed graph $G = (V, E)$, in order to find $SP(s, t)$, the algorithm starts at the source vertex s, traverses other connected vertices in ascending order of distance. Dijkstra keeps a priority queue to get the minimal distance from s to all unselected vertices and uses a set S to store the vertices which have been selected. Initially, both S and the priority queue are empty. In every iteration, choose the unselected vertex $v \in V \backslash S$ with minimal distance $d(s, v)$ from the priority queue, add v into S, and store $d(s, v)$ as the shortest distance between s and v. For each neighbor w of v, update the distance $d(s, w) = \min\{d(s, w), d(s, v) + l(v, w)\}$. When the target vertex t is visited, $SP(s, t)$ has been found. The asymptotic time consumption of Dijkstra is $O(m + n \log m)$, where n is the number of nodes, and m is the number of edges.

Bi-Directional Dijkstra [29]. For query (s, t), in bi-directional Dijkstra, a forward search from s and a reverse search from t will start simultaneously. Both of them are Dijkstra searches but in opposite directions. Vertices are visited in ascending order of $r_{s,t}(v) = \min\{d(s, v), d(v, t)\}$. With these two searches, bi-directional Dijkstra is much more efficient than the traditional Dijkstra. Many state-of-the-art shortest path algorithms [30], [31] are based on bi-directional Dijkstra.

2.2 Contraction Hierarchies

Contraction hierarchies (CH) [30] is a preprocessing-based shortest path algorithm which exploits hierarchical index. In preprocessing, CH calculates distances between some pairwise vertices, then uses the results to accelerate the query. Concretely, CH contracts the vertices in bottom-up order and adds shortcuts to ensure the correctness of shortest distance

computation in the remained graph. The remained graphs in each iteration are denoted by $G_1, G_2, \ldots, G_n$. Suppose v is the next vertex to be contracted in G_i, and u is an incoming neighbor of v ($u \in N_{in}(v)$, where $N_{in}(v)$ is the set of all the incoming neighbors of v), w is an outgoing neighbor of v ($w \in N_{out}(v)$), for each $u \in N_{in}(v)$ and $w \in N_{out}(v)$, CH computes $d(u,w)$ after v is removed, and compares it with $l(u,v)+l(v,w)$, where $l(u,v)$ and $l(v,w)$ denote the lengths of the edges $u \to v$ and $v \to w$. If $l(u,v)+l(v,w) < d(u,w)$, $SP(u,w)$ must pass v, then a shortcut $u \to w$ should be inserted into G_{i+1}, whose length is $l(u,v)+l(v,w)$. With shortcuts, $d(u,w)$ calculated in G_{i+1} cannot be influenced by the removal of v. Vertices are sorted by the order of contraction, i.e., the rank of v, $r(v)=i$ iff v is contracted in the ith iteration. Then CH gets a total order of all vertices, where $r(u) > r(v)$ means u has higher rank than v. After preprocessing, the original graph G with shortcuts inserted turns into a new graph, denoted by G^*. In G^*, for any pair (s,t), if all vertices on $SP(s,t)$ (except s and t) have lower rank than s and t, i.e., if for all $v \in V(SP(s,t))$, $r(v) < r(s)$ and $r(v) < r(t)$, then there must exist a shortcut $s \to t$ with length $d(s,t)$.

Bi-directional Dijkstra finds $SP(s,t)$ by running searches from both s and t in contrary directions with CH pruning in G^*. Take the forward search from s as an example, only the edges (u,v) with $r(u) < r(v)$ will be visited.

The efficiency of CH is mainly determined by the total order. In practice, CH is a fast algorithm in most cases, but in the worst case, the complexity can reach up to $O(n^2 \log n)$.

2.3 Labeling

Labeling method is a branch of distance oracle, studied in [23], [32], [33], [34]. 2-hop labeling [23] attempts to find a subset of vertices with the best connectivity in the graph. Each vertex u has a label $L(u)$, which is regarded as a simplification of shortest distance computation. For directed graphs, $L(u)$ includes the forward label $L_f(u)$ and reverse label $L_r(u)$. $L_f(u)$ contains the information about the outgoing shortest paths from u and $L_r(u)$ contains the information about the incoming shortest paths to u. Based on 2-hop labeling, any reachability, shortest path or distance query (s,t) can be answered by just taking $L_f(s)$ and $L_r(t)$.

Hub label algorithm (HL) [22] is a practical implementation of the labeling strategy. In HL, both $L_f(u)$ and $L_r(u)$ consist of an array. Take $L_f(u)$ as an example, elements of the array are pairs like $(v, d(u,v))$, where vertex v is called a *hub*. A pair (u,v) is *covered* by hub w iff w is on one of the shortest paths between u and v. Labels obey *cover property*: for any two vertices s and t, there is at least one vertex w on $SP(s,t)$ in both $L_f(u)$ and $L_r(u)$.

Hierarchical hub labeling (HHL) [35], [36], [37] can be considered as HL with hierarchies of vertices. Hierarchical labeling satisfies following conditions:

(1) For any path $P(u,v)$, there exists a sole highest-rank vertex w on it.
(2) For any pair (u,v), the vertex with the highest rank on $SP(u,v)$ is in both $L_f(v)$ and $L_r(u)$.

Thus, on a shortest path $SP(s,t)$, the vertex with the highest rank can be found out in $L_f(s) \cap L_r(t)$. Taking advantage of this feature, we adopt labeling strategy in the preprocessing.

3 Problem Definition

A real world network can be considered as a directed weighted graph $G=(V,E)$, where $V=V(G)$ is the set of all the nodes, and $E=E(G)$ is the set of all the edges. There is a weight function $E \to R^+$ to map each edge to its weight. We use $n=|V|$ to denote the number of vertices and $m=|E|$ to denote the number of edges. The weight of edge e is denoted as $l(e)$.

For any $s,t \in V$, our sketch of $SP(s,t)$, which is denoted as $SP^*(s,t)$, consists of the top-k critical vertices in $SP(s,t)$, where k is a user-defined integer.

Definition 1 (Significance). *Every vertex v in G has a significance. Significance of v, denoted as $sg(v)$, can be regarded as the importance of v. Without loss of generality, we assume that for any $u,v \in V$, $sg(u) \neq sg(v)$. If $sg(s) > sg(t)$, significance of s is higher than t, which means s is more significant than t.*

Definition 2 (Top function). *$Top(S)$ stands for the highest-significance vertex in S, where S is a set of vertices.*

$$Top(S) = \arg\max_{v_x \in S} sg(v_x) \tag{1}$$

Similarly, for a path P, $Top(P)$ means the highest-significance vertex on P.

For a path $P(u,v)$, also denoted as P_{uv}, we define $Top^*(P_{uv})$ as the highest-significance non-endpoint vertex on P_{uv}:

$$Top^*(P_{uv}) = \arg\max_{v_x \in V(P_{uv}) \wedge v_x \neq u,v} sg(v_x) \tag{2}$$

Definition 3 (kCV query). *The input of top-k critical vertices (kCV) query is a quadruple (G,s,t,k), where k is the user-defined number of required critical vertices ($k>0$). The result is an array $((v_1, d(s,v_1)), (v_2, d(s,v_2)), \ldots, (v_k, d(s,v_k)))$. These kCV objects make up $SP^*(s,t) = v_{p_1} \to \cdots \to v_{p_k}$, where $p_1 \sim p_k$ are the orders of kCV objects sorted by their actual positions on $SP(s,t)$. If $|V(SP)| \leq k$, then SP^* is exactly the complete shortest path, in this case $SP^*(s,t) = SP(s,t)$. We mark the kCV objects as $ktop(s,t) = \{v_1, \ldots, v_k\}$, where $sg(v_1) > sg(v_2) > \cdots > sg(v_k)$.*

Our problem can be described straightforward: returning the top-k highest-significance vertices on the shortest path between the given pair.

In Table 1, we present all the symbols used in this paper.

The idea of significance is related to hierarchical strategy. CH [30] is an example which adopts hierarchical index. For a CH shortest path query (s,t), a bi-directional search runs from s and t with shortcuts, which is bottom-up, i.e., the search visits vertices from lower to higher significance on $SP(s,t)$ in G^*. However, our goal is contrary to CH. We need a top-down search to quickly get the top vertices on shortest paths. For simplicity of description, in following sections we assume that the shortest path is unique, but we can also handle the case of multiple shortest paths, which will be discussed in Section 4.3.3.

4 Centralized Solutions to kCV Query

4.1 Overview

On centralized platform, we propose three algorithms for kCV query, consisting of both exact and approximate

TABLE 1
Symbols

Variable & Function	Description
G	Input graph
$V(\cdot)$	The set of vertices on a path or graph
$E(\cdot)$	The set of edges on a path or graph
n	The number of nodes in G
m	The number of edges in G
$l(\cdot)$	Weight of edge
$N_{in}(\cdot)$	The incoming neighbors of a vertex
$N_{out}(\cdot)$	The outgoing neighbors of a vertex
$sg(\cdot)$	Significance of vertex
$SP(\cdot)$	Shortest path
k	The number of user-defined required critical vertices
$ktop(\cdot)$	The kCV objects on the shortest path
$SP^*(\cdot)$	The sketch of the shortest path
$Top(\cdot)$	The highest significance vertex in a set or a path
$Top^*(\cdot)$	The highest significance non-endpoint vertex on a path
$hop(\cdot)$	The number of hops on a path
$L_f(\cdot)$	Forward label of a vertex
$L_r(\cdot)$	Reverse label of a vertex
$SPT(\cdot)$	Shortest path tree rooted at a vertex
$\psi(\cdot)$	The function which returns the top vertex on a shortest path
$\psi'(\cdot)$	The function which returns the top vertex (not endpoint) on a shortest path

methods. Exact solutions include the basic and the pure-labeling methods. To further improve the query efficiency, we explore an approximate approach with multi-threading technique, which can also achieve high accuracy and dramatically enhance the efficiency. All these three methods follow similar intuitions: relying on the preprocessed distance oracle, $SP(s,t)$ is split into subpaths progressively, and we add the critical vertices found on the subpaths into $ktop(s,t)$ in top-down order. We will give the proof for the correctness of our algorithms.

4.2 Basic Method

4.2.1 Preprocessing

We adopt hierarchical hub labeling [35] for the preprocessing to accelerate kCV query. The preprocessing consists of two parts: vertex ranking and label construction. We rank vertices with some predefined measure criteria of significance to obtain a total order of all $v \in V$, then we construct hierarchical labels for each vertex, including the forward label $L_f(v)$ and reverse label $L_r(v)$, ensuring that for any pair (s,t), $Top(SP(s,t))$ is in both $L_f(s)$ and $L_r(t)$. The online query is based on this offline preprocessing.

Vertex Ranking. In researches on the topology of networks, measuring the significance of vertices is fundamental. For kCV query, ranking vertices by their significance is quite essential for our subsequent algorithms because kCV query focuses on the significance of vertices, i.e., our query is order-sensitive. Denote the rank of vertex v as $r(v)$, where $(r(v) \in \{0,1,\ldots,n-1\})$. A vertex u is the lowest-significance vertex iff $r(u) = 0$.

We list some typical criteria for vertex ranking on networks as follow:

(1) Degree: the degree (including in-degree and out-degree) of v is the number of edges incident to v. Vertices with higher degree tend to be more significant in most cases.

(2) Betweenness centrality (betweenness): this is a common and representative standard to measure the significance of vertices, applied widely in networks. Betweenness centrality of vertex v can be computed by this expression:

$$betweenness(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3}$$

where σ_{st} is the number of the shortest paths from vertex s to vertex t, $\sigma_{st}(v)$ is the number of the shortest paths from s to t which pass v.

(3) Stress centrality: the number of the shortest paths that pass through the vertex.

$$stress(v) = \sum_{s \neq v \neq t} \sigma_{st}(v) \tag{4}$$

If we keep a shortest path tree $SPT(v)$ for every $v \in V$ (if the shortest paths are not unique, keep a DAG), and denote the set of the descendants of w (including w) in $SPT(v)$ as $descendants(w, SPT(v))$, then $stress(w)$ can be estimated by:

$$stress(w) \approx \sum_{v \in V} |descendants(w, SPT(v))| \tag{5}$$

In label-based algorithms, the ranking criterion is a main factor to determine the label size. Consequently, it has a marked impact on query efficiency and space consumption. The ranking criterion we apply should: 1) evaluate the vertices in a reasonable and meaningful way, i.e., the criterion cannot be set randomly or aimlessly, instead, it must reflect realistic meanings for the users; 2) try to improve the label quality and reduce the time and space consumption in query. Fortunately, researches [30], [36], [38] have shown that the two requirements have common ground and comparability in practice. Although finding a rank for optimal HHL is an NP-complete problem [39], there are some polynomial-time approximation algorithms [35], [36] for smallest labeling. In our implementation, we adopt the ranking methods in [30], [36], first use the preprocessing of CH to contract the low-significance vertices and reduce the size of G to get a much smaller remained graph G' containing only the top-L vertices, then reorder these vertices with selected criteria. Finally, combine the reordered top-L vertices and the rest ones to get the total order.

Label Construction. With a given total order, labels should be generated for each vertex. A naive method is running Dijkstra from every vertex u and putting $(u, d(u,v))$ into $L(v)$ for each $v \in SPT(u)$. A much more efficient way is pruning labeling (PL) [38], which is similar to Dijsktra, but can efficiently establish hierarchical labels with pruning. The main idea is: the initial labels are all empty; process vertices in top-down order (from the highest to the lowest significance); in every iteration, add a vertex into relevant labels. Specifically, when vertex u is processed, run two pruned Dijkstra:

(1) Start at vertex u, run forward Dijkstra (only visiting outgoing edges), then for every vertex v visited by the search, run HL query on current partial labels to compute an estimated value $d_h(u,v)$ for $d(u,v)$. If $d_h(u,v) \leq d(u,v)$, that means (u,v) has been covered by previous hubs, thus the search can be pruned by ignoring v and v's descendants. Otherwise, insert $(u, d(u,v))$ into $L_r(v)$ and continue the search.

(2) Same as above, but in the opposite direction. Run reverse search (only visiting incoming edges) to construct forward labels.

After applying this algorithm on every vertex, hierarchical labels have been constructed in a top-down way.

4.2.2 Query

We first introduce several utility methods which help design the basic method, then we present the whole process and analyze the characteristics of this algorithm.

ψ Operation. For any pair (u,v), we traverse $L_f(u) \cap L_r(v)$ and find a subset V', where each $v_i \in V'$ can minimize $d(u,v_i)+d(v_i,v)$. Notice that $Top(V') = Top(SP(u,v))$, we select $v_x = Top(V')$ and denote this operation as:

$$\psi(u,v) = Top(\{v_i \in L_f(u) \cap L_r(v) | \arg\min_{v_i}(d(u,v_i)+d(v_i,v))\}) \quad (6)$$

Operation $\psi(u,v)$ can be considered as a function which can return $Top(SP(u,v))$.

Assistant Query. Assistant query aims to find the neighbors of a vertex v_x on the shortest path which passes through v_x. To achieve it, we can exploit any distance oracle which supports high-efficiency shortest distance query. $N_r(v_x,u)$ is defined as the set of all incoming neighbors of v_x on $SP(u,v_x)$. By symmetry, $N_f(v_x,v)$ is the set of all outgoing neighbors of v_x on $SP(v_x,v)$. Take $SP(u,v_x)$ as an example, for every incoming neighbor v_m of v_x, we use an efficient sub query on pair (u,v_m) to find $d(u,v_m)$. If $d(u,v_m)+l(v_m,v_x) = d(u,v_x)$, it means that $v_m \in N_r(v_x,u)$. $v_n \in N_f(v_x,v)$ is obtained in a similar way. Since HHL query [35] is an efficient way to get the shortest distance, we can adopt it in the assistant query method.

Some relevant definitions are listed as follows:

Definition 4 (Strict partition). *Suppose path P is split into subpaths $P_1, P_2, \ldots, P_x$, if $V(P) = V(P_1) \cup V(P_2) \cup \cdots \cup V(P_x)$, and for any $1 \leq i \neq j \leq x$, $E(P_i) \cap E(P_j) = \emptyset$, then P is strictly partitioned by these subpaths. For vertex pairs, (s,t) is strictly partitioned by $(s_1,t_1),(s_2,t_2),\ldots,(s_x,t_x)$ iff there exist shortest paths $P(s,t), P(s_i,t_i), i=1,\ldots,x$, where $P(s,t)$ is strictly partitioned by $P(s_1,t_1), P(s_2,t_2), \ldots, P(s_x,t_x)$.*

The pseudo code of the basic method is presented in Algorithm 1. The query method is a divide-and-conquer process. $SP(s,t)$ is split into many subpaths progressively. The algorithm starts with an empty array $ktop = \emptyset$ (line 1), $\psi(s,t)$ returns the 1st *candidate* vertex (line 2). We use a priority queue to store the candidate vertices, where the elements in the queue are served according to their significance (from higher to lower). In the ith iteration, the priority queue pops out the highest-significance unselected candidate vertex x, we select x as the ith CV object (lines 4-5). Then, the subpath

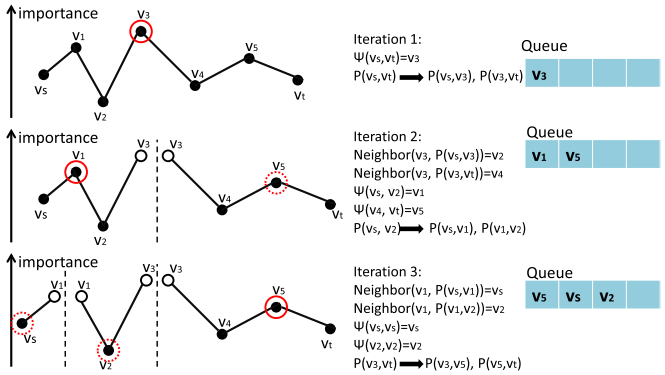


Fig. 2. An example of the basic method.

(u,v) which passes through x is split into two ones by x (lines 6-8). Obviously x is the top vertex on both $P(u,x)$ and $P(x,v)$, thus ψ operations are meaningless on these two subpaths. Instead, we use assistant queries to find the neighbors $m \in N_r(x,u)$ (lines 9-16) and $n \in N_f(x,v)$ (lines 17-24). If the shortest paths are not unique, we break ties in favor of the higher-significance vertices. After that, $P(u,v)$ is split into $P(u,m)$ and $P(n,v)$ (line 25). In a nutshell, the new subpaths which are split by last selected CV object are denoted as the *active* subpaths (Initially, $SP(s,t)$ is regarded as an active path). We do ψ operations on the active subpaths and add the top vertices on them into the priority queue as new candidate vertices (lines 13-14, 21-22). The algorithm runs iteratively and terminates when it returns all the kCV objects.

Algorithm 1. Basic Method of kCV Query

```
1: ktop = ∅
2: queue.push(ψ(s,t))
3: while |ktop| < k and !queue.isEmpty do
4:    x=queue.pop()
5:    ktop.push(x)
6:    Subpath sp = x.getsubpath()
7:    Vertex u = sp.getStart()
8:    Vertex v = sp.getEnd()
9:    Distance d_ux = findDistance(u,x)
10:   for m ∈ N_in(x) do
11:      Distance d_um = findDistance(u,m)
12:      if (d_um + l(m,x) = d_ux) then
13:         Vertex top_um = ψ(u,m)
14:         queue.push(top_um)
15:      end if
16:   end for
17:   Distance d_xv = findDistance(x,v)
18:   for n ∈ N_out(x) do
19:      Distance d_nv = findDistance(n,v)
20:      if (d_nv + l(x,n) = d_xv) then
21:         Vertex top_nv = ψ(n,v)
22:         queue.push(top_nv)
23:      end if
24:   end for
25:   splitPath(sp,m,n)
26: end while
```

Fig. 2 shows an example of the basic method to answer a 3CV query (v_s,v_t). ψ operation on $SP(v_s,v_t)$ returns the 1st CV object v_3 (in full-line red circle). With assistant queries, we find $v_2 \in N_r(v_3,v_s)$ and $v_4 \in N_f(v_3,v_t)$, then split $SP(v_s,v_t)$

into two active subpaths $SP(v_s, v_2)$ and $SP(v_4, v_t)$. In the 2nd iteration, candidate vertices v_1 and v_5 (in dotted-line red circles) are found and stored into the priority queue. The queue pops out v_1 as the 2nd CV object. The process stops when the 3CV objects v_3, v_1, v_5 are all returned.

4.2.3 Analysis on Basic Method

Denote the set of all the shortest paths between (s, t) as $ASP(s, t)$. We have following analysis on the basic method:

Lemma 1. *With labels established in the way introduced in Section 4.2.1, the vertex returned from operation ψ is the highest-significance vertex on all shortest paths of the given pair.*

Proof. As introduced in Section 4.2.1, we construct labels in top-down order, therefore, the higher-significance vertices are put into the labels earlier and cannot be pruned later, thus the top vertex on all shortest paths in $ASP(s, t)$ must be in both $L_f(s)$ and $L_r(t)$. □

We can imply Theorem 1 from Lemma 1:

Theorem 1 (Correctness of Basic Method). *The answer of the basic method for kCV query (s, t) is exactly the kCV objects on $SP_i(s, t) \in ASP(s, t)$, where $SP_i(s, t)$ is the shortest path which breaks ties in favor of the higher-significance vertices.*

Proof. If the shortest path $SP(s, t)$ is unique, considering that we retrieve the kCV objects in top-down order, the top k selected vertices are exactly the kCV objects. If the shortest paths are not unique, suppose in the gth iteration, the top $g-1$ CV objects $v_1, v_2, \ldots, v_{g-1}$ have been selected, and all current subpaths are $P(s_1, t_1), P(s_2, t_2), \ldots, P(s_x, t_x)$ (the subpaths may not be unique), notice that (s, t) is strictly partitioned by the top $g-1$ CV objects (consider them as $g-1$ pairs of the same vertex) and the pairs (s_1, t_1), $(s_2, t_2), \ldots, (s_x, t_x)$, thus the gth CV object must be on one of the subpaths. Denote the set of all current candidate vertices as $cdd(s, t)$. Suppose $c_i = \psi(s_i, t_i)$, notice that $cdd(s, t) = \{c_i | i = 1, \ldots, x\}$. By Lemma 1, c_i is the highest-significance vertex on all $SP_j(s_i, t_i) \in ASP(s_i, t_i)$. Therefore, $v_g = Top(\{c_1, c_2, \ldots, c_x\})$ has higher significance than any vertices which will be selected as critical vertices after v_g, i.e., v_g is exactly the gth CV object. In this way we can break ties in favor of the higher-significance vertices, and the vertices returned by Algorithm 1 are the exact kCV objects. □

Complexity. In the basic method, k determines the number of iterations. In each iteration, the time cost in each assistant query is decided by the average label size $|L|$, the number of assistant queries hinges on the average degree d, and the time consumption of the ψ operations depends on $|L|$. In total, the time complexity is $O(dk|L|)$. Compared with the algorithms which search the complete shortest path, we adopt offline preprocessing to reduce the online query latency by directly fetching the kCV objects. According to the recent work [35], $|L|$ can be surprisingly small even for very large graphs (about 69 for the road network of western Europe), thus each iteration can be finished in nearly constant time, which leads to the high efficiency of kCV query. As for the space consumption in preprocessing, with some compression methods [40], space cost can be dramatically reduced (only 0.8 GB for western Europe).

4.3 Pure-Labeling Method

In the basic method, in order to get the forward and reverse neighbors of the newly selected vertex on the shortest path, we need extra assistant queries. When the kCV query is finished, the total number of assistant queries reaches $O(dk)$.

Although those assistant queries are very efficient, if we can avoid them, the performance of kCV query can be better improved. The reason for using extra assistant queries is: for any subpath (u, v), $Top^*(SP(u, v))$ cannot be found directly. If labels have such a property: for any pair (u, v), $Top^*(SP(u, v))$ can be found in $L_f(u) \cap L_r(v)$, then the assistant queries are not necessary any more. Denote operation $\psi'(u, v) = Top^*(SP(u, v))$, all we need is applying $\psi'(u, v)$. We call this optimization as pure-labeling method.

4.3.1 Preprocessing

We use the same vertex ranking criterion as the basic method, but modify the way of label construction in the preprocessing of pure-labeling method.

If at least one shortest path between pair (s, t) passes through vertex v, it is defined as *v covers (s, t)*. As a further extension, we give a definition of *real-cover*:

Definition 5 (Real-cover). *If a vertex v is on at least one shortest path $SP_i(s, t) \in ASP(s, t)$, and $v \neq s, t$, then v real-covers (s, t).*

Initially, all labels are empty. We process the vertices in decreasing order of their significance. Suppose the next vertex to be processed is u, then we run Dijkstra search from it. The pruning principle is modified from the basic method. In the basic method, for every vertex v visited by the search, if $d_h(u, v) \leq d(u, v)$, then all the vertices in $descendants(v, SPT(u))$ can be pruned from $SPT(u)$. However, $d_h(u, v) \leq d(u, v)$ can only prove that $v_x = Top(SP(u, v))$ has been put into $L_f(u)$ and $L_r(v)$, but v_x might be an endpoint, i.e., $v_x = u$ or $v_x = v$. Thus we modify the principle of pruning while running the Dijkstra search from every $u \in V$ as below (take the forward search as example):

(1) If $d_h(u, v) < d(u, v)$, then (u, v) must have been covered by previous hubs except v, then we prune the $descendant(v, SPT(u))$ directly;
(2) If $d_h(u, v) > d(u, v)$, insert $(u, d(u, v))$ into $L_r(v)$ and continue the search as normal;
(3) If $d_h(u, v) \geq d(u, v)$ and $Top(X) = v$, where X is a set of vertices:

$$X = \{x | x = \underset{w \in L_f(u) \cap L_r(v)}{\arg\min} (d_h(u, w) + d_h(w, v))\} \quad (7)$$

then we add $(u, d(u, v))$ into $L_r(v)$, then $descendant(v, SPT(u))$ can be pruned.

Lemma 2. *With the preprocessing of pure-labeling, for any pair (s, t), $top^*(s, t) \in L_f(s) \cap L_r(t)$.*

Proof. For any pair (s, t), suppose $v = top(s, t)$ and $v^* = top^*(s, t)$. If $v = v^*$, obviously $v \in L_f(s) \cap L_r(t)$. Otherwise, notice that v^* is the highest-significance vertex in $V(SP(s, v^*))\backslash s$ and $V(SP(v^*, t))\backslash t$, thus, in the preprocessing, the Dijkstra search rooted by v^* can visit s and t, and then v^* is added into $L_f(s)$ and $L_r(t)$. Therefore, $top^*(s, t)$ can always be found in $L_f(s) \cap L_r(t)$. □

4.3.2 Query

The pseudo code of the pure-labeling method is shown in Algorithm 2. In pure-labeling method, by omitting assistant queries, the implementation can be much more concise: just do ψ' operations to select the kCV objects (lines 10-13) and split subpaths directly (line 9) in each iteration. The correctness of the answer of pure-labeling method can be proved similarly to the basic method.

Algorithm 2. Pure-Labeling Method of kCV Query

1: $ktop = \emptyset$
2: $queue$.push($\psi'(s,t)$)
3: **while** $|ktop| < k$ and !$queue$.isEmpty **do**
4: x=$queue$.pop()
5: $ktop$.push(x)
6: Subpath $sp = x$.getsubpath()
7: Vertex $u = sp$.getStart()
8: Vertex $v = sp$.getEnd()
9: splitPath(sp)
10: Vertex top_{ux}=$\psi'(u,x)$
11: Vertex top_{xv}=$\psi'(x,v)$
12: $queue$.push(top_{ux})
13: $queue$.push(top_{xv})
14: **end while**

Complexity. Inferred from Lemma 2, the assistant queries are not necessary in pure-labeling method, thus the time complexity can be reduced to $O(k|L|)$.

4.3.3 Multiple Shortest Paths

Since shortest paths are barely unique in most graphs, several strategies are presented to deal with this case.

- Determine which path to be chosen with some criteria. For example, break ties online in favor of higher significance vertices, as we applied in Section 4.2.2.
- Return the kCV objects on every shortest path $SP_i(s,t) \in ASP(s,t)$.

For the second strategy, we need some modification based on the pure-labeling method. First, in the preprocessing, while running the Dijkstra search from every $u \in V$, for each v visited by the search, the principle of pruning is (take the forward search as example):

(1) If $d_h(u,v) < d(u,v)$, prune $descendant(v, SPT(u))$ directly;
(2) If $d_h(u,v) = d(u,v)$, then we add $(u, d(u,v))$ into $L_r(v)$, and $descendant(v, SPT(u))$ can be pruned;
(3) If $d_h(u,v) > d(u,v)$, insert $(u, d(u,v))$ into $L_r(v)$ and continue the search as normal.

Lemma 3. *With labels established in above way, for each shortest path $SP_i(s,t) \in ASP(s,t)$, $L_f(s) \cap L_r(t)$ must contain $Top(SP_i(s,t))$.*

Proof. Similar to Lemma 1, but notice that for every $v_{sp_i} = Top(SP_i(s,t))$, where $SP_i(s,t) \in ASP(s,t)$, if we split $SP_i(s,t)$ into $SP_i(s, v_{sp_i})$ and $SP_i(v_{sp_i}, t)$, apparently v_{sp_i} is the highest-significance vertex on both $SP_i(s, v_{sp_i})$ and $SP_i(v_{sp_i}, t)$, according to the new pruning strategy, v_{sp_i} must be put into $L_f(s)$ and $L_r(t)$. Therefore, the top vertices on each shortest path $SP_i(s,t) \in ASP(s,t)$ are in both $L_f(s)$ and $L_r(t)$. □

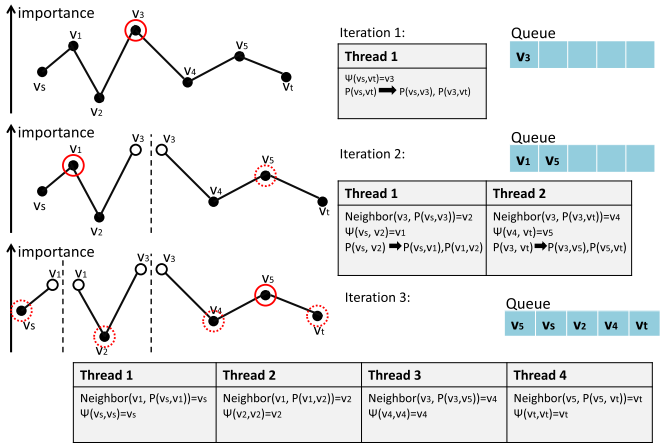


Fig. 3. An example of the multi-threading method.

In query, while splitting a subpath (u,v), we use a set T_m to store all the vertices which minimize $d(u,v_x) + d(v_x,v)$ for $v_x \in L_f(u) \cap L_r(v)$, i.e., $T_m = \{v_x \in L_f(u) \cap L_r(v) | \arg\min_{v_x} (d(u,v_x) + d(v_x,v))\}$. For each $v_i \in T_m$, use assistant queries to collect all the neighbors in $N_r(v_i,u)$ and $N_f(v_i,v)$, then continue the same process in Algorithm 2. If we denote the average number of the shortest paths between the same pairs as $|N_p|$, the complexity only needs to be multiplied by it as $O(dk|L||N_p|)$. Observing that $|N_p|$ can be considered as a small constant in real-world networks, the complexity can be still regarded as $O(dk|L|)$.

4.4 Multi-Threading Method

In order to further improve the performance of the previous methods, we switch to the approximate method and implement it in parallel with multi-threading technique. Although the basic method is already reasonably fast for real-time applications, we can still explore a parallel algorithm to achieve higher efficiency with a little sacrifice of optimality.

In the ith iteration, the above sequential methods split only one subpath (the subpath which passes the ith CV object). To enhance the performance, we propose a parallel method which has the same preprocessing as the basic method but differs in query:

In an iteration, suppose $SP(s,t)$ has been split into x subpaths: $P_1, P_2, \ldots, P_x$ (critical vertices selected in earlier iterations have been removed from $SP(s,t)$), do ψ operations on $P_1 \sim P_x$ in parallel, and get the corresponding candidate vertices $v_1 \sim v_x$, then put them into a priority queue which supports the multi-threading technique. When the number of all selected candidate vertices reaches αk, where α is a positive parameter ($\alpha \geq 1$), the algorithm terminates. The αk vertices are sorted by significance and the top-k ones of them are returned as result.

An example is shown in Fig. 3 where we set $\alpha = 1.5$ and $k = 3$. For query (v_s, v_t), in the 2nd iteration, $P(v_s, v_t)$ has been split into two subpaths $P(v_s, v_2)$ and $P(v_4, v_t)$. The relevant operations on them run in parallel, then the candidate vertices v_1, v_5 are pushed into a priority queue. In the 3th iteration, four subpaths are processed simultaneously, then the total number of all candidate vertices is $|\{v_3, v_1, v_5,$

$v_s, v_2, v_4, v_t\}| = 7 > \alpha k = 4.5$, thus the algorithm stops and returns the top-3 vertices v_3, v_1, v_5.

Speed Up. In each iteration of a sequential method, only one critical vertex will be selected and only one subpath will be split. Regard these as one *unit* operation, during the whole query process, totally there are k unit operations (suppose there are more than k hops on the shortest path). In the multi-threading method, in each iteration, unit operations on different subpaths can run simultaneously. In optimal case, suppose the algorithm runs fully in parallel, compared with the sequential methods, the number of iterations in multi-threading method can be logarithmically decreased to $\log(\alpha k)$, then the efficiency of multi-threading method exceeds the basic method by $k/\log(\alpha k)$ times. Considering that the number of threads in a computer is limited, the parallel method might not achieve logarithmical speedup, but according to our experimental results, the speedup of the multi-threading method is still quite notable.

Accuracy. As this algorithm splits paths in parallel, the result might not be the exact kCV objects. However, considering the complex factors in real-world networks, approximate results are still acceptable. The accuracy of the answer of multi-threading method is determined by the topology of the graph and the total order. We denote the kCV objects returned by the multi-threading method of kCV query (s,t) as $Mktop(s,t)$. The accuracy of the multi-threading method can be measured in different ways.

Definition 6 (Similarity). *Denote $com(s,t) = Mktop(s,t) \cap ktop(s,t)$. Similarity is defined as the quotient of $|com(s,t)|$ and the total number of the returned critical vertices.*

$$Similarity(s,t) = \frac{|com(s,t)|}{\min\{k, hop(SP(s,t))\}} \tag{8}$$

hop() denotes the number of hops on the path. In the best case, the exact kCV objects can all be found in query and the similarity is 1. In the worst case, every time a subpath is split, the rest critical vertices in $ktop(s,t)$ all lie in the same side, which determines the lower limit of similarity as $\log(\alpha k)/k$. However, the worst case hardly happens in real-world networks, and the experimental results of random queries show that the accuracy is quite high in practice.

We sort the vertices in $Mktop(s,t)$ and $ktop(s,t)$ respectively in decreasing order of significance. Suppose u is the ith vertex in $Mktop(s,t)$, and v is the jth vertex in $ktop(s,t)$, we mark i as the $index$ of u in $Mktop(s,t)$ (and j is the index of v in $ktop(s,t)$). We define $com_p(s,t)$ as the set of vertices which have same indices in both $Mktop(s,t)$ and $ktop(s,t)$.

Definition 7 (Pos-similarity). *We define pos-similarity as the quotient of $|com_p(s,t)|$ and the total number of returned critical vertices.*

$$Pos\text{-}similarity(s,t) = \frac{|com_p(s,t)|}{\min\{k, hop(SP(s,t))\}} \tag{9}$$

α serves to control the tradeoff between the speed and accuracy. In our experiment, the accuracy is quite good even when $\alpha = 1$.

5 Distributed Solutions to kCV Query

This section introduces the algorithm and implementation on distributed platform for kCV problem. On centralized platform, the algorithms require a high-performance computer with large memory, but the problem can be solved with a cluster which consists of a set of ordinary machines. Thus we study on applying our algorithm to distributed paradigms, which have been widely used in recent research work [41], [42], [43], [44]. We use Spark [45], a general and efficient distributed engine for large-scale data processing as our platform. Spark adopts resilient distributed dataset (RDD), which is a distributed collection of elements, supporting parallel operations and fault-tolerant mechanism.

The basic kCV query cannot be well parallelized, for the basic method is a sequential process. The observation that only those highest-significance vertices are needed in most cases leads to our current implementation on distributed platform, where we can use the most important landmarks to construct the labels and parallelize the query algorithm.

As distributed processing is more appealing to applications with heavy workload, single kCV query doesn't need a distributed platform to accelerate it, instead, we investigate a batch of kCV queries in distributed framework, which can process a batch of kCV queries by taking advantage of the distributed computing power. Top-k batch (kBCV) query is faced with multiple simultaneous requests on all pairs of vertices in a query set. kBCV query is useful in many cases. For example, in social network, query for the important individuals of the connections between every pair in a group is quite common. In web graphs, searching for the significant hubs on all-pair accesses in a LAN is also frequently applied. We implement the algorithm efficiently, exploiting reverse labeling and pruning strategy.

5.1 Preprocessing

Under most circumstances, only those most critical vertices are of the users' interest. With regard to this observation, we investigate approximate approaches, which only focus on those highest-significance vertices. In our implementation, vertices which have higher significance than a fixed value R_s are selected as *landmarks*. The set of landmarks is denoted by S_l, i.e., $R_s = |S_l|$. Labels are established only with the landmarks. Label construction on distributed platform is a multi-source shortest path search rooted by the $|S_l|$ landmarks. In the search from each $l_i \in S_l$, put l_i and the corresponding distance into the labels of all the vertices visited.

Traditional shortest path searches in large graphs are quite costly. In order to improve the performance of the preprocessing, we use *shortcuts* inserted by the local shortest path searches to reduce the search space in label construction. Labels are generated with the $|S_l|$ landmarks on the *auxiliary graph* G^+, which consists of the original graph G and those inserted shortcuts. With these shortcuts, labels can be constructed more efficiently.

As Algorithm 3 shows, the preprocessing consists of two steps. In the first step (lines 1-17), we run local shortest path searches from every $v \in V$ with a hop limit H^*, where H^* is a positive integer. For each local shortest path $SP_l(u,v)$ with $hop(SP_l(u,v)) = H$, insert a shortcut $u \to v$ with length $d_l(u,v)$ into G, where H is a positive integer and $H < H^*$,

$d_l(u,v)$ is the length of $SP_l(u,v)$. In real-world networks, with proper H chosen, nearly all local shortest distances can approach or be equal to the actual distances. The shortcuts make G into auxiliary graph G^+.

Algorithm 3. Preprocessing of kBCV Query with Landmark set S_l on Distributed Platform

```
1: for u, v ∈ V do
2:    distanceList(u, v) = ∞
3: end for
4: for u ∈ V do
5:    calculate d_l(u, v), v ∈ V with localBFS(u, H*)
6:    for all ((u, v), d_l(u, v)) do
7:       if d_l(u, v) < distanceList(u, v) then
8:          update distanceList with ((u, v), d_l(u, v))
9:       end if
10:   end for
11: end for
12: G+ = G
13: for (u, v) ∈ distanceList do
14:   for hop(SP_l(u, v)) = H do
15:      insert shortcut u → v with length d_l(u, v) into G+
16:   end for
17: end for
18: for v ∈ S_l do
19:   for all vertices u_i ∈ SPT(v) do
20:      update L(u_i) with (v, d(u_i, v))
21:   end for
22: end for
```

After shortcuts are inserted, we run a multi-source shortest path search in G^+ to establish the labels (lines 18-22). Shortcuts can reduce the diameter D of G (the largest number of hops on all shortest paths) to at most $\frac{D}{H}+H$. In this way the search space can be decreased and the performance is enhanced. We denote the labels generated with $|S_l|$ landmarks as $|S_l|$-$index$.

5.2 KBCV Query

With distributed framework, we can process a batch of kCV queries simultaneously. A formal definition is given as follows:

Definition 8 (kBCV query). *Given a set of vertices $S_q = \{v_1, v_2, \ldots, v_{|S_q|}\}$ and a positive integer k, a kBCV query returns the kCV objects on $SP(v_i, v_j)$ for every pair (v_i, v_j), $v_i, v_j \in S_q$.*

In our algorithm, labels are distributively stored in an RDD with ⟨key, value⟩ pairs as elements, where each vertex ID v_i is the key and the label $L(v_i)$ is its value. First the labels are reversed and the hubs are output as keys. As a pruning method, the hubs are sorted by significance, then only the top-$k\delta$ highest-significance hubs are selected as *candidate* critical vertices, where δ is a positive number ($\delta \geq 1$). The pruning method is based on the fact: the highest-significance vertices can cover most shortest paths. In most cases, critical vertices query can be solved with only a small set of those highest-significance hubs. With the pruning strategy, the algorithm provides approximate results, where parameter δ serves to control the tradeoff between accuracy and efficiency. The larger δ is, the fewer hubs are pruned, resulting in higher accuracy at the cost of time and space. Considering that the highest-significance vertices are of most users' interest, users could set a parameter R as a threshold to limit the significance of returned vertices. If users set R, those hubs with lower significance than R are also pruned. In this way the query only returns critical vertices with significance above R. With $|S_l|$-index, we can answer kBCV queries with any threshold $R \leq |S_l|$. The user-defined R can help the algorithm to control the pruning more flexibly and achieve higher efficiency. After pruning, reversed pairs will be distributed to multiple machines according to their hubs. We calculate the shortest distances and return the kCV objects for each (u,v), $u, v \in S_q$. By computing the shortest distance for each pair in parallel, kCV objects can be found efficiently. The process of computing is scalable with load balance strategies.

An implementation of our solution to kBCV query is shown in Algorithm 4. For convenience of description, we just display the case of undirected graphs, but it is simple to extend the method to directed case. Here we explain Algorithm 4 in detail. On each machine, for every $u \in L(v)$, we reverse the original pair from $\langle v, (u, d(u,v))\rangle$ to $\langle u, (v, d(u,v))\rangle$ (lines 1-3). The pruning is implemented with a *filter* operation on the RDD (lines 4-8). Then the shortest distances can be calculated simultaneously on different machines (lines 9-12). Shuffle operations are required for the repartition of data across machines. In the last step, kCV objects on every $SP(v_i, v_j)$ are returned (lines 13-15), where $v_i, v_j \in S_q$.

Algorithm 4. A Distributed Implementation of kBCV Query

```
1: for each element (u, d(u, v)) ∈ L(v) where v ∈ S_q do
2:    Output ⟨u, (v, d(u, v))⟩
3: end for
4: for all ⟨u, (v, d(u, v))⟩ do
5:    filter out the hubs u, where r(u) < R
6:    select top k * δ hubs, denoted by u_1, u_2, ..., u_{k*δ}
7:    Output ⟨u_i, (v, d(u_i, v))⟩, i ∈ [1, k * δ]
8: end for
9: for each ⟨u, (v_i, d(v_i, u))⟩ and ⟨u, (v_j, d(u, v_j))⟩ do
10:   d(v_i, v_j) = d(v_i, u) + d(u, v_j)
11:   Output ⟨(v_i, v_j), (u, d(v_i, v_j))⟩
12: end for
13: for all ⟨key, value⟩ with key=(v_i, v_j) do
14:   Output⟨(v_i, v_j), kCV objects on SP(v_i, v_j)⟩
15: end for
```

Complexity. Suppose there are p machines on the cluster. If $p = \infty$, the algorithm is fully parallelized. Reversing labels in step 1 takes $O(|S_q||S_l|)$ time. Then a shuffle is required to select the top $k\delta$ candidate critical vertices. Time consumption in distance computation decreases with p. For each candidate vertex, computing the distance costs $|S_q|^2$. In total, the time complexity is $O(|S_q||S_l|/p + |S_q|^2/p + T_s)$, where T_s denotes the time cost in shuffle. The space consumption is $O(n|S_l|)$.

6 Experiments

In this section we will present experimental results of the algorithms mentioned in this paper. The graphs we tested

TABLE 2
Instances

Graph	Network type	Nodes	Edges	Directed	weighted	Average degree	Max degree
NewYork	Road	264,346	733,846	undirected	weighted	2.78	16
BAY	Road	321,270	800,172	undirected	weighted	2.49	14
Florida	Road	1,070,376	2,712,798	undirected	weighted	2.53	16
NWUSA	Road	1,207,945	2,840,208	undirected	weighted	2.35	18
CAL	Road	1,890,815	4,657,742	undirected	weighted	2.46	14
EUSA	Road	3,598,623	8,778,114	undirected	weighted	2.44	18
CUSA	Road	14,081,816	34,292,496	undirected	weighted	2.44	18
Brighkite	Social	58,228	214,078	undirected	unweighted	7.35	2268
WikiTalk	Social	2,394,385	5,021,410	directed	unweighted	2.10	100032
Orkut	Social	3,072,441	117,185,083	undirected	unweighted	38.14	33313
NotreDame	Web	325,729	1,497,134	directed	unweighted	4.60	10721
Google	Web	875,713	5,105,039	directed	unweighted	5.83	6353

TABLE 3
Performance of Preprocessing with Different Ranking Strategies

Instances	Degree			$PageRank$			RK_1			RK_2		
	L_a	Prep(s)	Space(GB)	L_a	Prep(s)	Space(GB)	L_a	Prep(s)	Space(GB)	L_a	Prep(s)	Space(GB)
NewYork	610.9	365.2	1.22	685.1	486	1.36	88.9	64.5	0.2	68.7	468	0.16
BAY	784.3	656.8	1.89	1121.8	1202.3	2.70	69.4	35.8	0.19	49.9	404.2	0.14
Florida	1584.3	8764	12.68	2715.6	31127	21.7	87.1	133.3	0.79	67.2	448.6	0.62
NWUSA	804.2	2865	7.28	1907.4	17263	17.2	90.2	167.2	1.44	75.3	469.5	1.21
CAL	1179.4	9319.4	16.7	DNF	DNF	DNF	98	234.8	2.96	94	500	2.84
EUSA	DNF	DNF	DNF	DNF	DNF	DNF	139.4	649.1	9.16	127	989	8.29
CUSA	DNF	DNF	DNF	DNF	DNF	DNF	285.3	8142	38.7	230	9728	30.6
Brighkite	90.0	16.9	0.04	86.3	17.5	0.04	83.2	32.3	0.04	76.3	387	0.04
WikiTalk	67.9	1372.0	1.30	78.9	1300.9	1.49	64.3	2348.4	1.23	60.5	4348.5	1.16
NotreDame	21.1	22.5	0.07	33.9	50	0.11	20.3	53.7	0.07	18.6	78.4	0.07
Google	143.4	1173.9	1.01	161.5	1689.7	1.13	141.8	4326.2	1.00	137.0	6374	0.97

are real-world networks taken from the 9th DIMACS website[1] and Stanford large network dataset collection,[2] including road networks, social networks and web graphs. For each graph, Table 2 gives detailed characteristics.

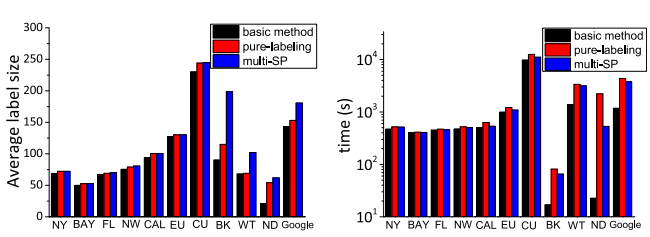
6.1 Experiments on Centralized Platform

On centralized platform, we implemented our algorithms in C++ and compiled them with Visual Studio 2013. Experiments were conducted in a 64-bit Windows 8 workstation with two Intel Xeon CPUs (3.33 GHz) and 64 GB memory. Each CPU has 4 cores. In the experiments of multi-threading method, all 8 cores were used with OpenMP by default. We adapted the methods of label construction in [30], [35], [36] for the preprocessing, with no label compression methods.

6.1.1 Preprocessing

For the basic method, Table 3 shows the experimental results of the preprocessing on all instances with different strategies of significance ranking. The performance indicators include the average label size (L_a), preprocessing time and memory space. We employ the preprocessing of CH [30] to remove the nonsignificant vertices, which are measured by the properties of them, including the degree and betweenness, then reorder the top L highest-significance vertices in the remained graph, following the way in [36]. We show the result of $RK_1(L = 0)$ and $RK_2(L = 15164)$, in comparison with the degree strategy and PageRank, a widely-applied ranking algorithm. For PageRank, we set the damping factor $d = 0.85$ and convergence criterion $\varepsilon = 0.00001$. From Table 3 we find that for social networks and web graphs, degree strategy and PageRank work well, but they perform worse than other strategies in road networks. Considering that in road networks, the degree of different vertices does not vary obviously, and the paths are much longer than other types of graphs, thus degree and PageRank are not suitable strategies for road networks. For them, betweenness is a better standard. With proper realistically meaningful vertex properties for different networks considered, the label sizes can be well reduced and be of benefit to the query efficiency.

Fig. 4 shows the preprocessing of the basic method, pure-labeling method and the case of multiple shortest paths (multi-SP) mentioned in Section 4.3.3, and compares them in terms of space and time consumption. For road networks,



(a) space (b) time

Fig. 4. Preprocessing of the basic method, pure-labeling and multi-SP.

1. http://www.dis.uniroma1.it/challenge9/
2. https://snap.stanford.edu/data/

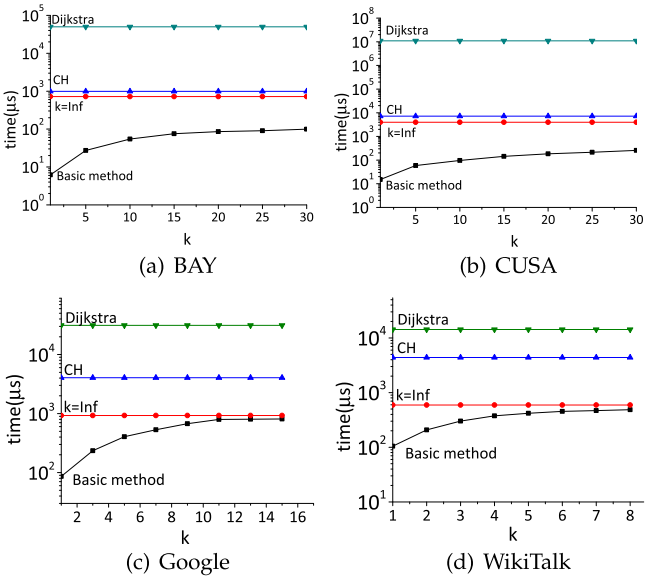


Fig. 5. Efficiency of basic method and traditional methods for kCV query.

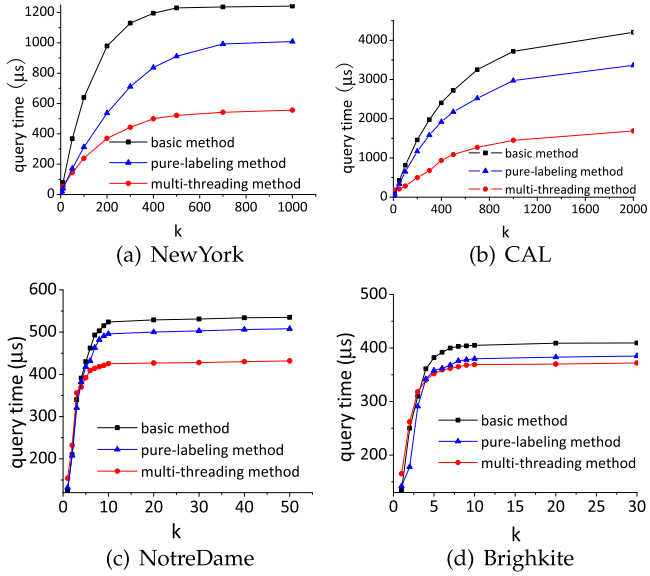


Fig. 6. Time consumption of the basic method, multi-threading method and pure-labeling method.

the results of these three methods do not differ obviously. The cost of pure-labeling preprocessing exceeds the consumption of the basic method by $1.5\% \sim 25.6\%$ in time and $2.3\% \sim 6.3\%$ in space. For multiple shortest paths, the time consumption is $0.7\% \sim 13.7\%$ more than the basic method, and the space cost is $2.9\% \sim 6.5\%$ higher. For social networks and web graphs, considering that there are many shortest paths between same pairs, the costs of pure-labeling and multi-SP methods also turn higher.

6.1.2 Efficiency of Different Methods of Query

Fig. 5 compares the efficiency of the basic method and some traditional shortest path algorithms to answer kCV query. The time consumption of the basic method increases with k. Obviously, the basic method outperforms other methods by $1.4 \sim 5$ orders of magnitude. Traditional algorithms cannot directly obtain the kCV objects in top-down order. Even with optimization, they still have to get much more than k vertices, or even all vertices on the shortest path and then compare their significance. We also present the time consumption when we set $k = \infty$, i.e., query for the complete shortest paths. The results prove that our basic method is an efficient way of shortest path sketch.

Fig. 6 presents the time consumption of the basic method, multi-threading method and pure-labeling method when k varies from 1 to D. Each time we randomly chose 1,000,000 pairs of vertices for kCV queries, then calculated the average query time.

As the overall trend shown in Fig. 6, for the general methods, the query time increases obviously (nearly linearly for sequential methods) with k when k is small. This is reasonable because k determines the number of unit operations, which dominates the time consumption if the label sizes of different vertices are assumed to be uniform. When k grows larger, for many query pairs, k approaches to or even exceeds the number of hops on their shortest paths. Therefore, the curves of query time tend to become smooth.

Fig. 6 compares the three methods in terms of the query time. In the experiments of multi-threading method, we set $\alpha = 1.0$ and use 8 threads. All the three methods can respond in microsecond level, competent for real-time kCV queries. In comparison, the time cost of the basic method is most expensive. Pure-labeling method outperforms it by $14\% \sim 51\%$. In road networks, the performance of multi-threading method is a bit subtle: when k is very small, its query latency is slightly higher than the basic method. This phenomenon is attributed to the cost of multi-threading technique. However, when k grows, multi-threading method shows its superiority, which outperforms pure-labeling method by $3 \sim 6$ times. In social networks and web graphs, most shortest paths have much less hops than road networks, for example, in Brightkite, most shortest paths consist of only $3 \sim 10$ vertices. In this case, the multi-threading method is not necessary. We can further combine the sequential and multi-threading methods to obtain a more flexible solution: apply the basic method if k is very small, and use multi-threading method to accelerate the query when k increases.

6.1.3 Effect of α on Accuracy and Efficiency

We investigate the effect of α in multi-threading method. The experimental results confirm our analysis in Section 4.4. Fig. 7 describes the change trend of query time with different α. Not surprisingly, time consumption increases with larger α. When $\alpha = 8$, the query latency is $1.5 \sim 2.3$ times higher than the time cost when $\alpha = 1$. Fig. 8 shows the accuracy with different values of $\alpha = 1.0, 1.5, 2, 3, 4, 8$.

In general, the results have deviations from the exact kCV objects, but they are also considerably accurate. For most instances, similarity stays over 95 percent and possimilarity always exceeds 70 percent when $\alpha \geq 4$. In fact, we find the quality of the result is already acceptable when $\alpha = 1$, achieving 70 percent similarity even in the worst case.

The curves in Fig. 8 are in the shape of check marks. Accuracy is high when k is small but decreases gradually and reaches a bottom when k grows. This is owing to the growing number of the subpaths which should be split. If k continues increasing, then the accuracy rises again because αk approaches to D, leading to more common vertices returned by both sequential and multi-threading methods.

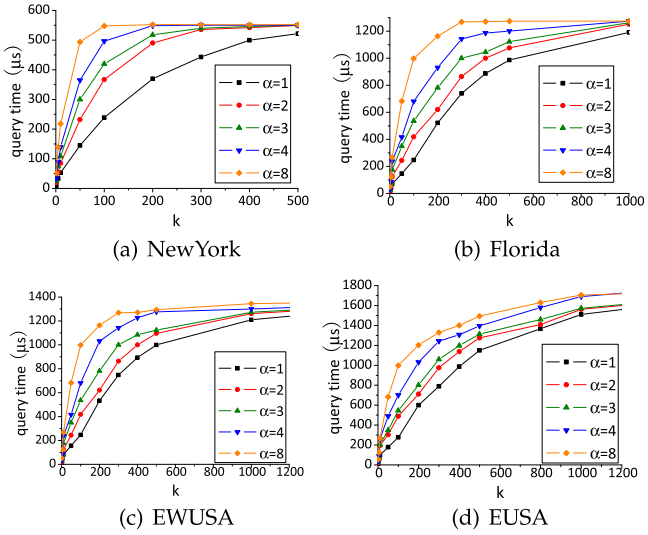


Fig. 7. Time consumption with different α in multi-threading method.

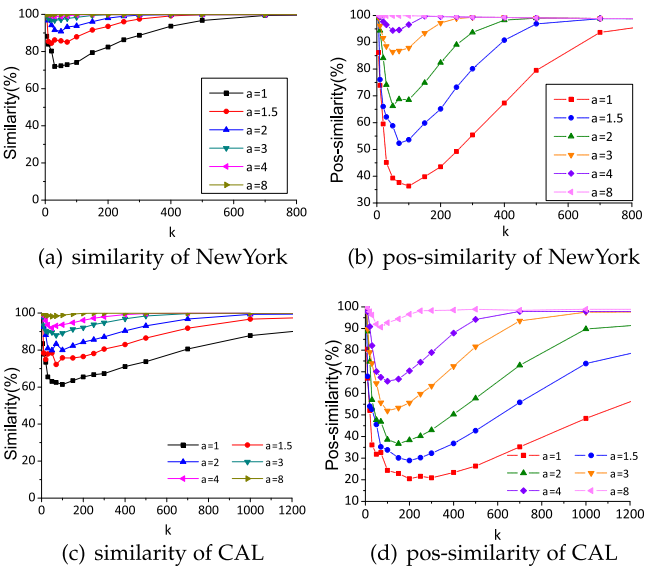


Fig. 8. Accuracy (similarity and pos-similarity) with different α in multi-threading method.

6.1.4 Parallelization in Multi-Threading Method

Fig. 9 shows the impact of parallelization on query latency when we set the number of threads N_t to 1, 2, 4, 8 respectively. From the figure we can imply that the speedup of multi-threading technique is not obvious when k is small, but if k is larger, the query performance is apparently improved when N_t increases. Every time when we double N_t, the query efficiency can be enhanced by $1.2 \sim 1.9$ times.

6.2 Experiments on Distributed Platform

We conducted experiments with Spark 1.3.0 in Scala, on a cluster with 9 workers, each worker has 6 cores and 18 GB memory, totally 54 cores and 162 GB memory. For each instance, we preprocessed indices with $R_s = 500$ sequentially. Unless otherwise stated, in each experiment on kBCV query, we randomly picked query sets, each consists of 1,000 vertices, set threshold $R = R_s$, then calculated the average results of the 1,000,000 kCV queries.

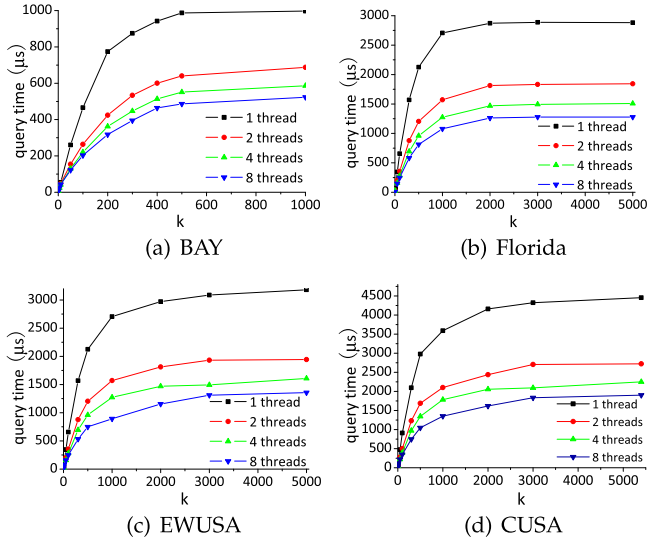


Fig. 9. Time consumption with different numbers of threads in multi-threading method.

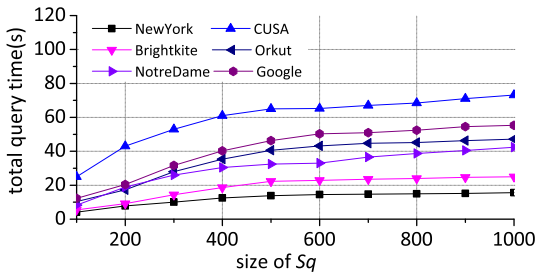


Fig. 10. Total time cost of kBCV query with different $|S_q|$.

6.2.1 Performance of kBCV Query with Varying $|S_q|$

Fig. 10 presents the total time of a kBCV query on S_q with different sizes. The figure reveals that kBCV query latency is sublinear with respect to $|S_q|$. With the optimization techniques of the distributed platform, kBCV queries with larger query sets perform better in average latency, which indicates that distributed platform is appealing to kBCV queries.

6.2.2 Efficiency of kBCV Query with Varying k and δ

The performance of kBCV query is shown in Fig. 11, where k varies from 1 to 32, δ is set to 1, 2, 4, 10, 15, 20, 25. For fixed δ, there is an obvious increasing tendency when k increases, which accords with the intuition. When we focus on the effect of δ, the figure indicates that higher δ incurs higher time consumption. When $\delta = 10$, average query time is $2 \sim 3.2$ times higher than the cost when $\delta = 1$. That is because higher δ leads to weaker pruning and higher workload of computation.

6.2.3 Accuracy of kBCV Query

Fig. 12a presents the accuracy of kBCV query when δ varies from 1 to 25 and k is fixed to 20. We measure the accuracy by similarity, i.e., the ratio of the common kCV objects between the answers returned by the basic method and Algorithm 4 to all the vertices returned by Algorithm 4. As δ serves to control the tradeoff between efficiency and accuracy, larger δ leads to higher accuracy at the cost of query time. On average, accuracy stays higher than 70% with $\delta \geq 10$ for most instances.

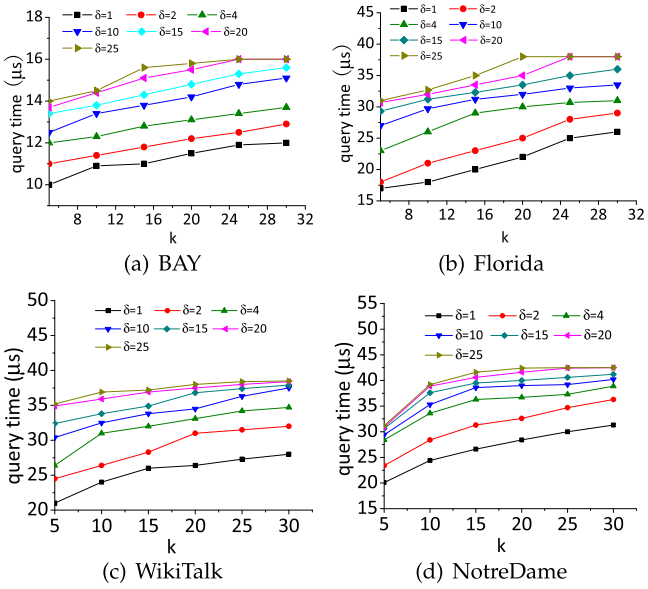


Fig. 11. Efficiency of kBCV query with different δ and k.

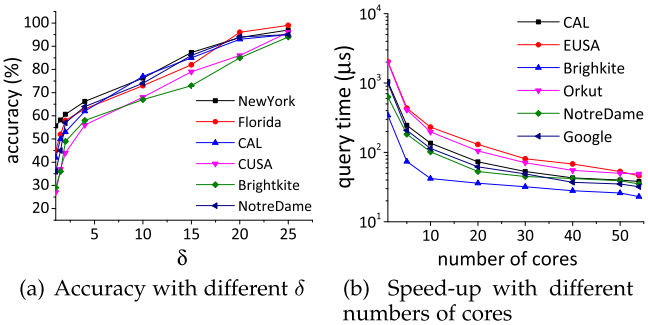


Fig. 12. Accuracy and speed-up of kBCV query.

6.2.4 Speed-Up of kBCV Query

On distributed platform, the speed-up with the increasing number of computation cores in the cluster is a main feature. Fig. 12b describes the time cost with different numbers of cores N_c used in the cluster. We set $k = 20$ and $\delta = 4$. In practice, most time is spent in shuffle, but with more cores in the cluster, computation is distributed and the total time can be substantially reduced. According to the figure, the efficiency is greatly enhanced with more cores. The average query time when $N_c = 54$ is improved by $13 \sim 32$ times compared with the case when $N_c = 1$. This result confirms the satisfying speed-up of kBCV query on distributed platform.

7 Conclusion

In this paper, we propose kCV query as a kind of shortest path sketch. Our algorithm is based on hierarchical strategies, applying distance oracle to deal with shortest distance queries in the preprocessing. As for the query, we stand on a new point of view, pay attention to those crucial vertices on the shortest path and study on algorithms on both centralized and distributed platforms. On centralized platform, we propose a basic algorithm as a fundamental solution and then extend it to other methods with optimization and parallelism. Multi-threading method is especially attractive, which can achieve much higher efficiency with slight sacrifice of accuracy.

On distributed platform, we investigate kBCV query, which aims to process a batch of kCV queries. By taking advantage of distributed computation, the algorithm is speeded up.

All methods of kCV query mentioned in this paper are of high efficiency, returning an accurate sketch of shortest paths in microseconds.

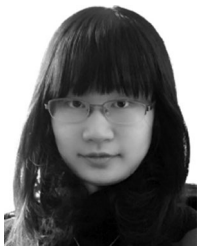
Acknowledgments

This work was supported by the NSFC (61729202, U1636210 and 61602297), the National Basic Research Program (973 Program, No. 2015CB352403), the National Key Research and Development Program of China (2016YFB0700502), the Scientific Innovation Act of STCSM (15JC1402400), the Opening Projects of State Key Laboratory of Software Development Environment (SKLSDE-2017KF-02), the Beijing Key Laboratory of Big Data Management and Analysis Methods, the Guizhou Provincial Key Laboratory of Public Big Data (2017BDKFJJ0), and Microsoft Research Asia and CCF-Tencent Open Research Fund RAGR20170114.

References

[1] Y. Tong, J. She, and R. Meng, "Bottleneck-aware arrangement over event-based social networks: The max-min approach," *World Wide Web*, vol. 19, no. 6, pp. 1151–1177, 2016.
[2] H. Huang, Y. Gao, L. Chen, R. Li, K. Chiew, and Q. He, "Browse with a social web directory," in *Proc. ACM SIGIR*, 2013, pp. 865–868.
[3] J. Zhao, Y. Gao, G. Chen, and R. Chen, "Towards efficient framework for time-aware spatial keyword queries on road networks," *ACM Trans. Inf. Syst.*, vol. 36, no. 3, 2017, Art. no. 24.
[4] Y. Gao, X. Miao, G. Chen, B. Zheng, D. Cai, and H. Cui, "On efficiently finding reverse k-nearest neighbors over uncertain graphs," *Int. J. Very Large Data Bases*, vol. 26, pp. 1–26, 2017.
[5] X. Miao, Y. Gao, G. Chen, B. Zheng, and H. Cui, "Processing incomplete k nearest neighbor search," *IEEE Trans. Fuzzy Syst.*, vol. 24, no. 6, pp. 1349–1363, Dec. 2016.
[6] Y. Gao, J. Zhao, B. Zheng, and G. Chen, "Efficient collective spatial keyword query processing on road networks," *IEEE Trans. Intell Transp. Syst.*, vol. 17, no. 2, pp. 469–480, 2016.
[7] Y. Gao, X. Qin, B. Zheng, and G. Chen, "Efficient reverse top-k boolean spatial keyword queries on road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 5, pp. 1205–1218, May 2015.
[8] D. Xie, G. Li, B. Yao, X. Wei, X. Xiao, Y. Gao, and M. Guo, "Practical private shortest path computation based on oblivious storage," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 361–372.
[9] F. Li, B. Yao, M. Tang, and M. Hadjieleftheriou, "Spatial approximate string search," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 6, pp. 1394–1409, Jun. 2013.
[10] B. Yao, F. Li, and X. Xiao, "Secure nearest neighbor revisited," in *Proc. 29th Int. Conf. Data Eng.*, 2013, pp. 733–744.
[11] B. Yao, F. Li, and P. Kumar, "K nearest neighbor queries and knn-joins in large relational databases (almost) for free," in *Proc. 26th Int Conf Data Eng.*, 2010, pp. 4–15.
[12] X. Xiao, B. Yao, and F. Li, "Optimal location queries in road network databases," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 804–815.
[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
[14] H. Bast, S. Funke, and D. Matijević, "TRANSIT: Ultrafast shortest-path queries with linear-time preprocessing," in *Proc. 9th DIMACS Implementation Challenge—Shortest Path*, 2006.
[15] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *Proc. Meeting Algorithm Eng. Exp.*, 2007, pp. 46–59.
[16] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: Towards bridging theory and practice," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 857–868.
[17] A. D. Zhu, X. Xiao, S. Wang, and W. Lin, "Efficient single-source shortest path and distance queries on large graphs," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 988–1006.

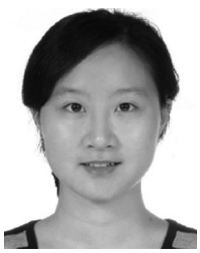
[18] S. Holzer and R. Wattenhofer, "Optimal distributed all pairs shortest paths and applications," in *Proc. ACM Symp. Principles Distrib. Comput.*, 2012, pp. 355–364.
[19] E. Solomonik, A. Buluc, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *Proc. IEEE 27th Int. Symp. Parall. Distrib. Process.*, 2013, pp. 548–559.
[20] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou, "Shortest path and distance queries on road networks: An experimental evaluation," *Proc. VLDB Endowment*, vol. 5, no. 5, pp. 406–417, 2012.
[21] Y. Tao, C. Sheng, and J. Pei, "On k-skip shortest paths," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 421–432.
[22] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "A hub-based labeling algorithm for shortest paths in road networks," in *Proc. 10th Int Conf. Exp. Algorithms*, 2011, pp. 230–241.
[23] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, pp. 1338–1355, 2003.
[24] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson, "Distributed computation of large-scale graph problems," in *Proc. 26th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2015, pp. 391–410.
[25] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A performance evaluation of open source graph databases," in *Proc. 1st Workshop Parallel Program. Analytics Appl.*, 2014, pp. 11–18.
[26] C. McCubbin, B. Perozzi, A. Levine, and A. Rahman, "Finding the 'needle': Locating interesting nodes using the k-shortest paths algorithm in mapreduce," in *Proc. IEEE 11th Int. Conf. Data Mining Workshops*, 2011, pp. 180–187.
[27] D. Nanongkai, "Distributed approximation algorithms for weighted shortest paths," in *Proc. 46th Annu. ACM Symp. Theory Comput.*, 2014, pp. 565–573.
[28] Z. Qi, Y. Xiao, B. Shao, and H. Wang, "Toward a distance oracle for billion-node graphs," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 61–72, 2013.
[29] T. A. J. Nicholson, "Finding the shortest route between two points in a network," *Comput. J.*, vol. 9, no. 3, pp. 275–280, 1966.
[30] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Proc. 7th Int. Conf. Exp. Algorithms*, 2008, pp. 319–333.
[31] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for a*: Shortest path algorithms with preprocessing," *Shortest Path Problem: 9th DIMACS Implementation Challenge*, vol. 74, pp. 93–139, 2009.
[32] S. Abbasi and S. Ebrahimnejad, "Finding the shortest path in dynamic network using labeling algorithm," *Int. J. Business Soc. Sci.*, vol. 2, no. 20, 2011.
[33] I. Abraham, A. Fiat, et al., "Highway dimension, shortest paths, and provably efficient algorithms," in *Proc. 21st Annu. ACM-SIAM Symp. Discrete Algorithms*, 2010, pp. 782–793.
[34] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: A high-compression indexing scheme for reachability query," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 813–826.
[35] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *Proc. 20th Annu. Eur. Conf. Algorithms*, 2012, pp. 24–35.
[36] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Robust distance queries on massive networks," in *Proc. Eur. Symp. Algorithms*, 2014, pp. 321–333.
[37] A. V. Goldberg, I. Razenshteyn, and R. Savchenko, "Separating hierarchical and general hub labelings," in *Proc. Int Symp. Math. Found. Comput. Sci.*, 2013, pp. 469–479.
[38] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proc. ACM SIGMOD Int. Conf. Manage Data*, 2013, pp. 349–360.
[39] M. Babenko, A. V. Goldberg, H. Kaplan, R. Savchenko, and M. Weller, "On the complexity of hub labeling," in *Proc. Int. Symp. Math. Found. Comput. Sci.*, 2015, pp. 62–74.
[40] D. Delling, A. V. Goldberg, and R. F. Werneck, "Hub label compression," in *Proc. Int. Symp. Exp. Algorithms*, 2013, pp. 18–29.
[41] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proc. Int. Conf. Manage Data*, 2016, pp. 1071–1085.
[42] G. Chen, K. Yang, L. Chen, Y. Gao, B. Zheng, and C. Chen, "Metric similarity joins using mapreduce," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 3, pp. 656–669, Mar. 2017.
[43] Y. Tong, X. Zhang, and L. Chen, "Tracking frequent items over distributed probabilistic data," *World Wide Web*, vol. 19, no. 4, pp. 579–604, 2016.
[44] L. Chen, Y. Gao, Z. Xing, C. S. Jensen, and G. Chen, "I2RS: A distributed geo-textual image retrieval and recommendation system," *Proc VLDB Endowment*, vol. 8, no. 12, pp. 1884–1887, 2015.
[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw Syst. Des. Implementation*, 2012, Art. no. 2.



Jing Ma is working toward the master's degree in the Computer Science and Engineering Department, Shanghai Jiao Tong University. Her research interests include techniques of graph analysis, database, and distributed computing.

Bin Yao received the PhD degree in computer science from the Department of Computer Science, Florida State University, in 2011. He is an associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include management and indexing of large databases, query processing in spatial and multimedia databases, string and keyword search, and scalable data analytics. He is a member of the IEEE.

Xiaofeng Gao received the PhD degree from the University of Texas at Dallas, in 2010. She is an associate professor of computer science and engineering with Shanghai Jiao Tong University. Her research interests include data engineering, database management, wireless network, and optimization algorithms. She is a member of the IEEE.



Yanyan Shen received the PhD degree from the National University of Singapore, in 2015. She is a special associate research fellow of computer science and engineering with Shanghai Jiao Tong University. Her research interests include big data analytics and processing.

Minyi Guo received the PhD degree in information science from the University of Tsukuba, Japan, in 1998. He is the head in the Department of Computer Science and Engineering, and the director of the Embedded and Pervasive Computing Center, Shanghai Jiao Tong University. His research interests include parallel and distributed processing; parallelizing compilers; cloud computing; pervasive computing; software engineering, embedded systems; green computing; and wireless sensor networks. He is a fellow of the IEEE.