Reverse k-Ranks Queries on Large Graphs

Yuqiu Qian, Hui Li, Nikos Mamoulis, Yu Liu, David W. Cheung
Department of Computer Science
The University of Hong Kong, Hong Kong
{yqqian,hli2,nikos,yliu4,dcheung}@cs.hku.hk

ABSTRACT

Given a collection of objects, the reverse k-ranks query takes as input a query object q in the set and returns the top-k objects that rank q higher compared to where other objects rank q. This query has been studied in the vector space, however, there is no previous work in the context of graphs. In this paper, we propose a filter-and-refinement framework, which prunes the search space while traversing the graph in search for the reverse k-ranks query results. We present an optimized algorithm and an index that apply on this framework and boost its performance. The proposed techniques are evaluated on real data; the experimental results show that our solutions scale well, rendering the query applicable for searching large graphs.

1. INTRODUCTION

Ranking queries (e.g., k-NN query [10], reverse k-NN query [13], reverse top-k query [21], reverse k-ranks query [27]) have become very popular in database management systems. Among them, the reverse k-ranks query has been recently proposed as an enhancement of the reverse top-k query, which ensures the same number of results for any query input. Specifically, given a *customer-product* vector space, where customers rank products, the reverse top-k query takes as input a product and an integer k, and produces as output the k customers that rank the product higher compared to its ranking by other customers. However, there is no prior work on how to evaluate reverse k-ranks queries on graphs, where graph proximity measures can be used to define the distance between nodes (and their ranking with respect to a query node).

Motivation. The reverse top-k query was extended to apply on large graphs in [26, 25]. Given a query node and an integer k, this query retrieves all other nodes that have the query node in the set of their k nearest nodes, based on a proximity measure. We conducted an experiment, where we apply reverse top-k queries (using shortest weighted path as the proximity measure) on the DBLP author collaboration graph [12]. Each node in the graph corresponds to an author and two authors are connected by an edge if they have published at least one paper together. Edges are weighted to reflect the strength of the collaboration [17, 11]. The application of a reverse top-k query is to let the query author know what other authors are keen to collaborate with him/her. According to our experiments in Section 6.2, for a large percentage of query nodes, the reverse top-k query returns either very few or too many results. The fact that reverse top-k queries do not have a fixed number of results limits their utility, especially in applications such as graph based recommender systems (e.g., tag recommendation [7], friendship recommendation [14], product recommendation [19] and paper recommendation [16]), where making recommendations to "cold-start" users who are weakly connected to the rest of the data is an important issue. On the other hand, the reverse k-ranks query returns a result of fixed size (k) for any query node; hence, it is particularly useful for new nodes of the graph (e.g., new social network users) who have little influence to other nodes and for "hot" nodes, which have very high influence but still want to shortlist the nodes that they are most attracted to them.

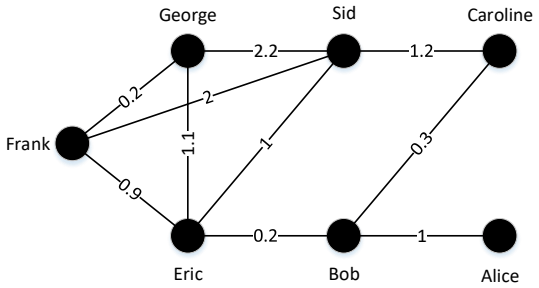


Figure 1: A Toy Example

EXAMPLE 1. *Figure 1 illustrates a toy example. Seven researchers form a weighted undirected graph. Alice is a new researcher and she only has a weak connection with Bob. If we use shortest path to measure proximity, we can get the rank matrix shown in Table 1. For example, Rank(Alice, Eric) is Eric's position in the list of nodes ordered by shortest path distance from Alice. Indeed, Eric is the 2nd closest node (after Bob) to Alice with a shortest path distance 1.2.*

A reverse top-k query having Alice as the query node with $k = 2$ returns no results since Alice does not fall into any researchers' top-2 list (See first column of Table 1). This means that this query is not useful to Alice for recommending her researchers to collaborate with. On the other hand, a reverse 2-ranks query for Alice returns a nonempty result {Bob, Caroline}, since both Bob and Caroline rank Alice higher compared to her rank by other researchers, which means that these are the two researchers who are most likely to collaborate with her. If the query node is Eric and we use a re-

verse top-2 query, we will recommend all other six researchers to him because of his close relationship to all of them; this result is overwhelming and would not be useful to Eric. On the other hand, a reverse 2-ranks query returns {Bob, Sid} (since Bob and Sid rank Eric as 1st while others rank him as 2nd).

Table 1: Rank Matrix

	Alice	Bob	Caroline	Sid	Eric	Frank	George
Alice	-	1	3	5	2	4	6
Bob	3	-	2	5	1	4	6
Caroline	4	1	-	3	2	5	6
Sid	6	2	2	-	1	4	5
Eric	6	1	2	4	-	3	5
Frank	6	3	4	5	2	-	1
George	6	3	4	5	2	1	-

Applications. In the era of big data, large volumes of graph data are becoming available. Reverse k-ranks queries over large-scale graphs can find application in spatial network data analysis, collaboration recommendation, dating, etc. For example, the management of a supermarket chain may want to investigate the space of potential customers. Given a road network which includes communities (i.e., estates) and supermarkets, a reverse k-ranks query will return a list of k communities which rank a given supermarket higher compared to where it is ranked by other communities, based on its network distance. The result can be used by the management to conduct targeted advertisement and promotion. As discussed, reverse k-ranks queries can also be used for collaboration and friendship recommendation in collaboration networks or social networks.

Contribution. In this paper, we study the evaluation of reverse k-ranks queries on large graphs, using shortest weighted path as the measure of proximity between nodes. To the best of our knowledge, there is no previous work on this problem. The special nature of graph data does not allow the application of the approaches proposed for reverse k-ranks queries in the vector space [27]. In addition, extending existing solutions for top-k search and reverse top-k search on graphs to compute reverse k-ranks queries is not trivial. Specifically, for a top-k search we only have to find the top-k proximity set of a single node q and for the reverse top-k search we need to compute the top-k sets of all nodes in the graph and check whether q appears in each of them. For the reverse k-ranks query, we must calculate all rank sets of all nodes in the graph and find the top-k ranks of q in them; therefore, a reverse k-ranks query is substantially more expensive than top-k search and reverse top-k search.

Our contributions can be summarized as follows:

- We study for the first time reverse k-ranks queries on large graphs and propose a filter-and-refine graph browsing framework to evaluate it. We propose effective bounds for the ranks of the examined nodes that limit the set of nodes which need to be accessed during query evaluation.
- We propose a dynamically refined, space-efficient index structure, which supports reverse k-ranks query evaluation. The index is paired with an efficient online query algorithm, which prunes a large number of nodes that are definitely in or not in the reverse k-ranks result and reduces the required refinements for the remaining candidates.
- We conduct an experimental study demonstrating the efficiency of our framework, as well as the effectiveness of the reverse k-ranks query in real graph applications.

The remainder of this paper is organized as follows. Section 2 provides a formal definition of reverse k-ranks search and discusses a baseline brute-force solution. In Section 3, we present a fundamental theorem and our basic two-step framework. Two efficient algorithms, Dynamic Bounded SDS-tree and Dynamic Bounded SDS-tree with index, are proposed in Section 4 and Section 5 respectively. Section 6 evaluates the effectiveness of reverse k-ranks queries and the efficiency of the proposed framework. In Section 7, we briefly discuss previous work related to reverse k-ranks queries. Finally, Section 8 concludes the paper.

2. PROBLEM DEFINITION

A formal definition of the reverse k-ranks query is given below:

DEFINITION 1. *(Rank(s,t)) Consider a weighted graph $G = (V, E)$, consisting of a set of nodes V and a set of edges E. Each edge in E carries a non-negative weight. For any two nodes $s, t \in V$, let $d(s,t)$ denote the shortest path distance from s to t, which is defined by summing up the weights of the edges along the shortest path from s to t. Let S be the set of nodes that satisfy $\forall p_i \in S, d(s,p_i) < d(s,t)$ and $\forall p_j \in (V - S - \{t\}), d(s,p_j) \geq d(s,t)$. Then, $Rank(s,t) = |S| + 1$ where $S \subset V$ and $|S|$ is the cardinality of S.*

DEFINITION 2. *(Reverse k-Ranks Query on a Graph) Given a weighted graph $G = (V, E)$, a query node q and a positive integer k, the reverse k-ranks query returns a subset T of V, such that $|T| = k$ and $\forall p_i \in T$, $\forall p_j \in (V - T - \{q\})$, $Rank(p_i, q) \leq Rank(p_j, q)$.*

Computing the reverse k-ranks set of a query node q is not trivial. A naive method, for each node $p_i \in V$, traverses the graph to find the distances to all other nodes from p_i in increasing order (i.e., using Dijkstra's algorithm) until q is encountered; this can give us Rank(p_i,q). During this process, the top-k of these ranks are maintained in a heap and eventually returned as results. Obviously, this method is very expensive. Another possible solution is to apply multiple reverse top-k' queries with an increasing k' value, until the number of results is similar to the k value of the reverse k-ranks query. This solution, apart from only giving an approximate result, is also expensive because the number of required reverse top-k' queries could be large and there is no straightforward method for evaluating them incrementally.

3. GENERAL TWO-STEP FRAMEWORK

To process reverse k-ranks queries efficiently, we design a two-step framework. First, we build a Shortest Distance Search tree based on the given query node and use it to prune the space of candidate nodes. Second, in a refinement step, we compute Rank(p_i,q) for each surviving candidate node p_i; during this process, the top-k nodes are maintained in a priority queue and they are finally output. Although our examples and illustrations are on undirected graphs, our solutions can directly be applied to directed graphs.

3.1 Filter step: SDS-Tree

Given a graph $G = (V, E)$, the Shortest Distance Search tree (SDS-tree) rooted at vertex q is a spanning tree T_q of graph G, such that the path distance from any other vertex p to q is the shortest path distance from p to q in G. SDS-Tree is similar to the Dijkstra tree [4], but on the transpose graph G^T, which can be different to G if G is directed. G^T is a directed graph on the same set of vertices as G, but with all of the edges of G reversed. That is, if G contains

Algorithm 1 Basic SDS-tree Construction

```
1: procedure REVERSEKRANK(q, G)
2:     Priority Queue Q ← {q : 0}                    ▷ nodes to visit
3:     R ← ∅                                         ▷ reverse k-ranks result
4:     D ← ∅                                         ▷ nodes visited
5:     kRank ← Inf                                   ▷ k-th top rank in R
6:     while Q do
7:         top ← Q.pop()
8:         D ← D ⋃ {top}
9:         top.rank ← GetRank(top, kRank)
10:        if top.rank ≠ −1 then                     ▷ Theorem 1
11:            Update R
12:            kRank ← new k-th rank in R
13:            for t in top.neighbors() do
14:                if t ∉ D then
15:                    dis ← top.dis + d[top][t]
16:                    if t ∈ Q and t.dis > dis then
17:                        t.dis ← dis
18:                    else
19:                        t.dis ← dis
20:                        Q.push(t)
21:    return R
```

an edge (u, v) then G^T contains an edge (v, u) with same weight and vice versa. If G is undirected, then $G^T = G$.

To build the SDS-tree for the query input node q, we run Dijkstra's algorithm on the reversed edges of the graph (Algorithm 1). Specifically, we maintain a priority queue of the current shortest distance from each node to query node q. Each time, we dequeue the node t with the shortest distance $d(t, q)$, add t to the tree by making it a child of its successor node in the shortest path from t to q, and update the distance from t's neighbors to query node q. At the same time we conduct a *rank refinement* for t; that is we compute $Rank(t, q)$. This rank refinement procedure (*GetRank* in Line 9 of Algorithm 1) will be explained shortly. During the tree construction process, every time we dequeue a node t and after updating $Rank(t, q)$, we maintain the set R of the nodes with the lowest $Rank(t, q)$ values (to be output at the end of the algorithm). The largest of the k lowest $Rank(t, q)$ values so far is denoted by $kRank$ and serves as a bound. The tree construction finishes when the shortest paths from all nodes to q have been determined.

For a large graph, constructing the entire SDS-tree is too expensive. We now show some nice properties of the SDS-tree that can help us to compute the reverse k-Ranks results, without having to build the whole tree.

LEMMA 1. *Consider a weighted graph $G = (V, E)$ and two nodes $p, q \in V$. For any node p' whose shortest path to q passes through p, Rank(p',q) ≥ Rank(p,q).*

PROOF. According to Definition 1, there must exist two sets S and T, such that Rank(p,q)=$|S| + 1$ and Rank(p',q)=$|T| + 1$. The aim here is to prove that $S \subset T$, which means that $|S| \leq |T|$. By definition, we know that $\forall p_i \in S$, $d(p, p_i) < d(p, q)$. Since all weights are non-negative, we further have $d(p', p) \geq 0$. Also, a path from p' to any p_i passes through p, which means that $d(p', p_i) \leq d(p', p) + d(p, p_i)$. As a result, we have $d(p', p_i) \leq d(p', p) + d(p, p_i) \leq d(p', p) + d(p, q) = d(p', q)$, therefore $\forall p_i \in S, p_i \in T$; i.e., $S \subset T$. □

Based on Lemma 1, we can easily obtain the following fundamental theorem:

THEOREM 1. *Given a SDS-tree T_q rooted at q and a node p of T_q, for any descendant p' of p, Rank(p',q) ≥ Rank(p,q).*

Based on Theorem 1, we can conclude that, given a SDS-tree T_q rooted at q, if node p is not in the reverse k-ranks query result of node q, then no child of p can be part of the result. This means that p's children need not be added to T_q during the tree construction; this can greatly limit the number of nodes p' that are added to the tree and for which $Rank(p', q)$ needs to be computed.

3.2 Rank Refinement

During the SDS-tree construction, for each node p that we visit and it is a candidate reverse k-ranks result, we have to apply a *rank refinement* procedure which computes $Rank(p, q)$. This is done by counting all nodes whose distance from p is shorter than $d(p, q)$. For this purpose, we build a *partial* Dijkstra tree starting from node p and we stop when we find q. The number of nodes that we encounter by this search is $Rank(p, q)$.

Recall that in Algorithm 1 we keep track of the set R of the lowest Rank values so far and of the current k-th top Rank value, denoted by $kRank$. During the refinement of $Rank(p, q)$, p can be pruned as soon as the number of nodes in the partial Dijkstra tree before reaching q is larger than $kRank - 1$, since in this case p has no potential to become a reverse k-ranks result, as well as its children nodes in the SDS-tree. The rank refinement step for a node is described by Algorithm 2.

Algorithm 2 Rank Refinement Algorithm

```
1: procedure GETRANK(node, kRank)
2:     Priority Queue Q ← {node:0}                   ▷ Nodes to visit
3:     D ← ∅                                         ▷ Nodes visited
4:     rank ← 1
5:     while Q do
6:         top ← Q.pop()
7:         D ← D ⋃ {top}
8:         for t in top.neighbors() do
9:             if t not in D then
10:                dis ← top.dis + d[top][t]
11:                if t ∈ Q and t.dis > dis then
12:                    t.dis ← dis
13:                else if t ∉ Q and node.dis > dis then
14:                    t.dis ← dis
15:                    Q.push(t)
16:                    rank ← rank + 1
17:                    if rank > kRank then
18:                        return −1                 ▷ Definition 2
19:    return rank
```

For the example of Figure 1, the SDS-tree is the same as the Dijkstra tree since G is an undirected graph (see Figure 2). Assume that k=2. The priority queue initially has 'Alice' as top element; after that, Algorithm 1 will perform rank refinement for Bob and get $Rank(Bob, Alice) = 3$. Since Bob can be in the reverse 2-ranks results of Alice, the neighbors of Bob (i.e. Caroline and Eric) are added to the priority queue Q. Because Eric's distance to Alice is shorter than that of Caroline, we will first do rank refinement of Eric, and get $Rank(Eric, Alice) = 6$. Then, the neighbors of Eric (i.e. Sid, George and Frank) will all be added to the priority queue. Then, since Caroline has the shortest distance to Alice, we will do rank refinement for it and get $Rank(Caroline, Alice) = 4$. After that, Frank, Sid and George will be rank-refined one by one.

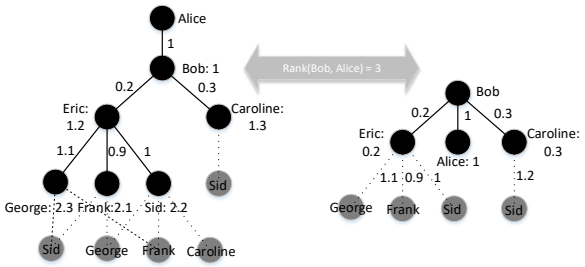


Figure 2: The SDS-tree for Example 1

4. DYNAMIC BOUNDED SDS-TREE

The filter-and-refinement framework described in the previous section significantly reduces the search space and it is much faster than the brute-force approach of computing the entire rank matrix. On the other hand, there may still be many false hits, i.e., nodes which are ranked-refined without ending up in the reverse k-ranks result. In Algorithm 1, each node is decided to be a candidate or not, immediately after refining its parent at the SDS-tree. However, at the time when a node p is dequeued, the current reverse k-ranks result (and the bound $kRank$) may have changed and it might be then possible to prune p just before its rank-refinement. We propose a Dynamic Bounded SDS-tree (DSDS-tree) approach, which is based on the idea of delaying the decision whether a node is a candidate to just before its rank refinement and on using a set of bounds to potentially prune the node.

In the DSDS-tree approach we maintain for each node p in the priority queue Q a lower bound of $Rank(p, q)$. p will be considered as a candidate right when it is dequeued; then, it is rank-refined only when its Rank lower bound is lower than the current k-th top Rank value (i.e. $kRank$). Only then p has a chance to enter the reverse k-Ranks query result. Specifically, we set the lower bound of $Rank(p, q)$ for each node p in DSDS-tree T_q rooted at node q as the maximum of the following three quantities: the depth of node p in tree T_q, the Rank value of its parent nodes, and the times visited so far during the rank-refinement of other nodes. Formally,

THEOREM 2. *Consider a DSDS-tree T_q rooted at query node q. For any node p whose depth is h and parent node is p', Rank(p,q)≥ max(h, Rank(p',q), $p.lcount$), where $p.lcount$ is the number of visited times for node p during the rank refinements of other nodes before actual refinement of node p itself.*

We can prove Theorem 2 by showing that each of the three quantities constitutes a lower bound by itself, therefore their maximum is a lower bound (the tightmost one, hence the most useful). In fact, we have already shown that $Rank(p', q)$ is a lower bound (Lemma 1). Now, we will prove the other two bounds.

We first demonstrate that $Rank(p, q)$ should not be less than the depth of node p in DSDS-tree T_q.

LEMMA 2. *Consider the DSDS-tree T_q of a query node q and suppose the depth for root is 0. For any node p whose depth is h, Rank(p,q)≥ h.*

PROOF. If p is at depth h, then the shortest path from p to q passes through $n = h-1$ nodes, i.e., the path is $\{p, p_1, p_2, ...p_n, q\}$. Since $\forall i \in [1, n], d(p, p_i) < d(p, q)$, we have $Rank(p, q) > n$, i.e., $Rank(p, q) \geq h$. □

We next show that $Rank(p, q)$ should not be less than the times that node p was visited during the rank-refinements of other nodes, before being refined itself.

LEMMA 3. *Consider a weighted graph $G = (V, E)$ and two nodes $p_1, p_2 \in V$, such that $d(p_1, q) \leq d(p_2, q)$. If $d(p_1, p_2) < d(p_1, q)$ holds, $d(p_2, p_1) < d(p_2, q)$ also holds.*[1]

PROOF. $d(p_2, p_1) = d(p_1, p_2) < d(p_1, q) \leq d(p_2, q)$, so $d(p_2, p_1) < d(p_2, q)$ holds. □

LEMMA 4. *Given a DSDS-tree T_q for a query node q, for any node p which has been visited during the rank refinements of other nodes for $p.lcount$ times before refinement of node p itself, Rank(p,q)≥ $p.lcount$.*

PROOF. For any node p, let T be the set of nodes which have been visited during the refinement of any node $p' \in T$ before node p itself, where $|T| = p.lcount$. This means that $d(p', q) \leq d(p, q)$ and $d(p', p) < d(p', q)$. Based on Lemma 3, we can conclude that $d(p, p') < d(p, q)$, which means that Rank(p,q)≥ $p.lcount$. □

In order to use Theorem 2, while building the *dynamic* SDS-Tree rooted at node q, we also need to maintain a priority queue of the current shortest distances from each node to query node q. Each time, we dequeue the node t with the shortest distance $d(t, q)$, we add t to the tree by making it a child of its successor node in the shortest path from t to q, and update the distance from t's neighbors to query node q if t successfully entered the current reverse k-ranks result set (as in Section 3). However, unlike the static SDS-tree, where for all nodes maintained in the priority queue we perform their rank refinement when we visit them, the dequeued nodes in the dynamic SDS-Tree are only rank-refined if their rank lower bound is smaller than the current $kRank$.

Consider the example in Figure 1. Similar to the basic framework of Section 3, the priority queue will initially have 'Alice' as root first. After dequeuing Alice and adding her neighbors in the queue, we will dequeue and rank-refine Bob to get $Rank(Bob, Alice) = 3$. Then, the neighbors of Bob (i.e. Caroline and Eric) will enter the priority queue. The rank refinement of Eric follows, giving $Rank(Eric, Alice) = 6$. Then, neighbors of Eric (i.e. Sid, George and Frank) will all enter the priority queue. Next, we will do the rank refinement of Caroline and get $Rank(Caroline, Alice) = 4$. The process can terminate here, since the lower bounds of ranks for Frank, Sid and Gorge are already larger than $kRank$. As a comparison, note that we would still have to do rank refinement for Frank, Sid and Gorge in the basic framework.

The lower-bound of Rank for each node can be dynamically updated during rank refinement steps. When meeting node t in the rank refinement of node p, we can update $t.lcount$ by adding 1, which can be done in constant time using a hash table. The whole space complexity will be $\mathcal{O}(|V|)$, but in practice we need much less space as the framework does not visit the nodes which are far from q.

5. INDEX-BASED SEARCH

In this section we propose an indexing approach which, when paired with the method presented in Section 4, can help to further reduce the cost of reverse k-ranks queries. A naive solution in this direction would be to precompute the entire rank matrix of size $|V| * |V|$. Starting from each node u, we can run a single-source shortest-path (SSSP) algorithm, i.e., build the entire Dijkstra tree, which can order all other nodes v by increasing $Rank(u, v)$ value. After computing the rank matrix, for each node v, we can sort the corresponding column of the matrix and obtain a ranked list of all

[1]This lemma holds for undirected graphs only. Therefore the count-based bound is not used in the case of directed graphs.

Algorithm 3 Build Dynamic SDS-Tree with Index Algorithm

```
1: procedure REVERSEKRANK(q, G)
2:     Priority Queue Q ← {q : 0}                          ▷ Nodes to visit
3:     R ← top-k in reverse_rank_dict                      ▷ result so far
4:     D ← ∅                                               ▷ Nodes visited
5:     kRank ← k-th top Rank in R
6:     while Q do
7:         top ← Q.pop()
8:         D ← D ⋃ {top}
9:         if top ∈ R then
10:            for t in top.neighbours() do
11:                if t ∉ D then
12:                    dis ← top.dis + d[top][t]
13:                    if t ∈ Q and t.dis > dis then
14:                        t.dis ← dis
15:                    else
16:                        t.dis ← dis
17:                        Q.push(t)
18:        LBound ← max(top.height, top.parent.rank, top.lcount, check_dic[top])
19:        if LBound ≥ kRank then
20:            continue
21:        top.rank ← GetRank(top, kRank)
22:        if top.rank ≠ −1 then                           ▷ Theorem 1
23:            Update R
24:            kRank ← new k-th rank in R
25:            for t in top.neighbours() do
26:                if t ∉ D then
27:                    dis ← top.dis + d[top][t]
28:                    if t ∈ Q and t.dis > dis then
29:                        t.dis ← dis
30:                    else
31:                        t.dis ← dis
32:                        Q.push(t)
33:    return R
```

other nodes u with respect to $Rank(u, v)$. For any reverse k-ranks query q, we can then return as results the first k nodes in the ranked list of q. The problem of this naive solution is that it is too expensive to precompute the entire rank matrix for very large graphs. Instead, we propose to only select a subset of H nodes, called *hubs*, and only run M iterations of SSSP from each hub node s to obtain s's top-M list of nearest nodes. For each node $t_i, i = \{1, 2, ..., M\}$ in this list, we simply know that $Rank(s, t_i)$ is the order of t_i in the list. The index also includes two additional components: a *Check Dictionary* and a *Reverse Rank Dictionary*, to be explained in Section 5.2. The index can facilitate the evaluation of reverse k-ranks, for k values not exceeding a parameter K. Index parameters H, M, and K are defined based on a precomputation cost/ speedup tradeoff. The larger these values are the more time it takes to create and maintain the index; on the other hand, reverse k-ranks queries are evaluated faster. In Section 6 we study the overhead and effectiveness of the index for various values of these parameters.

In the following, we first describe strategies for selecting the hub nodes and then show how we can use the precomputed information to initialize the index that can help to accelerate the computation of reverse k-ranks queries.

5.1 Hub Selection

We propose the following three strategies for selecting the hubs: random, degree first, closeness first. These strategies are experimentally evaluated in Section 6.

Random: We select the hubs randomly; this is used as a baseline to show the significance of other strategies.

Degree First: We select the vertices with the highest out-degrees as hubs. The reasoning behind this strategy is that vertices with higher out-degree have higher chances to connect with short shortest paths to many other vertices and therefore they have higher probability to be reverse k-Ranks query results.

Closeness First: We select as hubs the vertices with the highest closeness centrality. If we define farness of a node v as the sum of its distances from all other nodes, then closeness centrality is defined as the reciprocal of farness, which is $C(v) = 1/\sum_u d(u, v)$ [2, 18]. Since computing exact closeness centrality for all vertices requires $\mathcal{O}(|V| \cdot |E|)$ time [3] even for sparse graphs, we approximate closeness centrality by randomly sampling a small number of vertices and computing distances from those vertices to all vertices as in [1].

Algorithm 4 Dynamic Rank Refinement with Index Algorithm

```
1: procedure GETRANK(node, kRank)
2:     Priority Queue Q ← {node:0}                         ▷ Nodes to visit
3:     D ← ∅                                               ▷ Nodes visited
4:     rank ← 1
5:     while Q do
6:         top ← Q.pop()
7:         D ← D ⋃ {top}
8:         Update reverse_rank_dict
9:         for t in top.neighbours() do
10:            if t not in D then
11:                dis ← top.dis + d[top][t]
12:                if t ∈ Q and t.dis > dis then
13:                    t.dis ← dis
14:                else if t ∉ Q and node.dis > dis then
15:                    t.dis ← dis
16:                    Q.push(t)
17:                    rank ← rank + 1
18:                    t.lcount ← t.lcount + 1
19:                    if rank > kRank then
20:                        check_dic[node] ← D.size()
21:                        return −1                       ▷ Definition 2
22:    check_dic[node] ← rank
23:    return rank
```

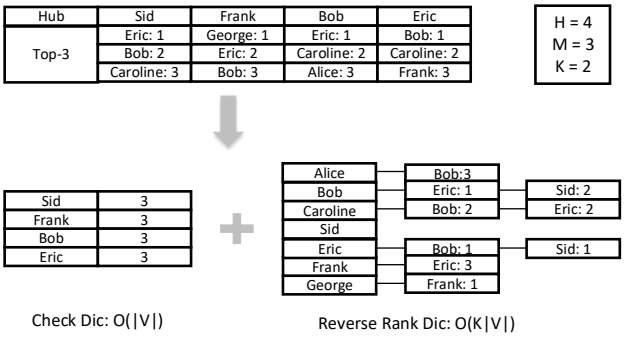


Figure 3: Index

5.2 Index Creation

After selecting H hubs following one of the heuristic strategies mentioned above, we build the index by running the SSSP algo-

rithm from each hub node u and stopping after obtaining the M nearest nodes from u. For each hub we store the list of M nearest nodes with their ranks. The index has two components: 1) the *Check Dictionary*; a hash-map, having nodes as keys and the number of steps SSSP has taken starting from these nodes as values (i.e., it contains an entry $\{u : M\}$, for each hub node u at the beginning); and 2) the *Reverse Rank Dictionary*; a set of adjacency lists, each corresponding to a node v and containing the current reverse K-ranks result of v ordered by rank value from small to large (recall that K is the maximum possible value of k).

Following the running example, suppose we choose {Sid, Frank, Bob, Eric} as the hubs, and precompute their top-3 rank list (i.e., $H = 4$ and $M = 3$). This gives us the rank matrix shown at the top of Figure 3. Assume that the largest possible value for k is $K = 2$. Then, we can set up the index with the following two parts. The *Check Dictionary* is {Sid:3, Frank:3, Bob:3, Eric:3} since from these four hub nodes we have so far retrieved the 3 nearest neighbors. The *Reverse Rank Dictionary* stores the existing reverse K-ranks result list for each hub node. Consider, for example, hub node Bob; we already know three Rank values, i.e. $Rank(Eric, Bob) = 1$, $Rank(Sid, Bob) = 2$ and $Rank(Frank, Bob) = 3$, but we only need to store the top-2 ranks in the Reverse Rank Dictionary, i.e., $Rank(Eric, Bob) = 1$ and $Rank(Sid, Bob) = 2$.

5.3 Querying and Index Updates

The proposed index is dynamic and can be updated whenever a new reverse k-ranks query is evaluated. For a new query node q, we first look up the *Reverse Rank Dictionary* to get any existing reverse k-ranks query results for q, which can give us an estimation for the k-th top Rank value (i.e. $kRank$). However, what we get from the index may not be the final query result; we may have to conduct more graph exploration. For this, we follow the general two-step framework described in the previous sections: we build the dynamic bounded SDS-tree, and do rank-refinement for candidate nodes. The index allows us to have a better estimation of the rank value $Rank(u, q)$ for each candidate node u: if u is in the *Reverse Rank Dictionary* of q, there is no need to do rank refinement for node u; if u is not in the *Reverse Rank Dictionary* of q, but the value of node u in the *Check Dictionary* is no smaller than the current $kRank$ value, there is also no need to do rank refinement for node u (i.e., u can be pruned). Otherwise, we have to do rank refinement for node u. For this purpose, we conduct SSSP search from u. During SSSP search, until the rank value of the nodes that we visit exceeds *Check Dictionary*$[u]$, we have to update *Reverse Rank Dictionary*. Specifically, if we reach node v with $Rank(u, v) = t_1$, and t_1 is smaller than the highest rank value of *Reverse Rank Dictionary*$[v]$, then we have to update *Reverse Rank Dictionary*$[v]$ with $\{u : t_1\}$. After finishing the rank refinement step from node u with $Rank(u, v) = t_2$, we also need to update the *Check Dictionary* with $\{u : t_2\}$. Algorithms 3 and 4 show how the search algorithm and its rank refinement module are adapted to use the index.

Following the previous example, we select {Sid, Frank, Bob, Eric} as the hubs, and precompute their top-3 ranks list as initial index. Consider Alice as the query. The index will be updated as shown in Figure 4. The index initially is as shown in Figure 3. The first step is to do rank refinement for Bob. However, as we can see in the index, the *Reverse Rank Dictionary* of Alice has {Bob:3}, which means that we need not update or compute anything and we can just turn to Eric directly. During the rank-refinement step of Eric, we also get the rank of other nodes for node Eric, and we can update the *Check Dictionary* and *Reverse Rank Dictionary* correspondingly: We add {Eric: 4} for Sid, {Eric: 5} for George, and finally {Eric: 6} for Alice, which is also the query node. After we reach Alice starting from Eric with Rank equals to 6, we also update the *Check Dictionary* with {Eric: 6}. Continuing this way, we proceed to rank-refine the next node (Caroline), and terminate, after updating again the *Check Dictionary* and the *Reverse Rank Dictionary*. Even though index updates incur extra costs (compared to the algorithm presented in Section 4), the updated index can help to speed up processing of future reverse k-ranks queries, as we demonstrate in the next section.

The space complexity for *Check Dictionary* and *Reverse Rank Dictionary* is $\mathcal{O}|V|$) and $\mathcal{O}(K \cdot |V|)$, respectively. The overall space complexity for both index componets is $\mathcal{O}(K \cdot |V|)$. The time complexity of building the index is reduced from $\mathcal{O}(|V| \cdot (|V| + |E| \cdot \log |V|))$ of the whole matrix building to $\mathcal{O}(H \cdot (M + |E^*| \cdot \log M))$ now, where $|E^*|$ is the number of edges linked with M nodes, estimated as $|E^*| = M \cdot |E|/|V|$ and bounded by $\mathcal{O}(|E|)$.

6. EXPERIMENTAL EVALUATION

In this section, we conduct an experimental evaluation for the effectiveness of reverse k-ranks queries on large graphs and verify the efficiency of our proposed algorithms. All tested methods were implemented in C++ and the experiments were conducted on a Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz machine, with 1 TB of main memory.

6.1 Datasets

Datasets DBLP, Epinions and SF are used in our experiments. General statistics of the three datasets are shown in Table 2.

Table 2: Data Sets

	DBLP	Epinions	SF
# of Nodes	1,314,050	75,879	321,678
# of Edges	18,986,618	508,837	800,172
Average Degree	14.45	6.71	2.49

Dataset DBLP[2] contains the collaboration graph of DBLP in May 2015. There are 1,314,050 nodes and 18,986,618 edges in this dataset. Each node in the graph denotes an author and authors who have collaborated are linked by edges. The edge weight between two nodes u and v is set to 1 divided by the number of co-authored papers by u and v increased by $\log_2 deg(u) + \log_2 deg(v)$ with normalization, where $deg(u)$ is the degree of node u [17, 11]. Setting the edge weights like this can produce less ties in the result set, which is important for unambiguous ranking quality evaluation.

Dataset Epinions[3] includes an online social network from the trust based reputation system Epinions.com. There are 75,879 nodes and 508,837 edges in this directed graph. Each node is a user of the system. One user can indicate whether he/she 'trust' another user's review (i.e., whether it is useful to him/her) and one trust statement forms an edge from the declarer to the target. Edge weights are sampled from a Zipf distribution with a skewness parameter $\alpha = 2$, as in [23].

Dataset SF contains locations of stores and road network information in San Francisco bay area. There are 408 stores in this dataset, which are crawled from GeoDeg[4]. The road network was made public during the 9th DIMACS Implementation Challenge[5].

[2]http://konect.uni-koblenz.de/networks/dblp_coauthor
[3]http://snap.stanford.edu/data/soc-Epinions1.html
[4]http://geodeg.com
[5]http://www.dis.uniroma1.it/challenge9/download.shtml

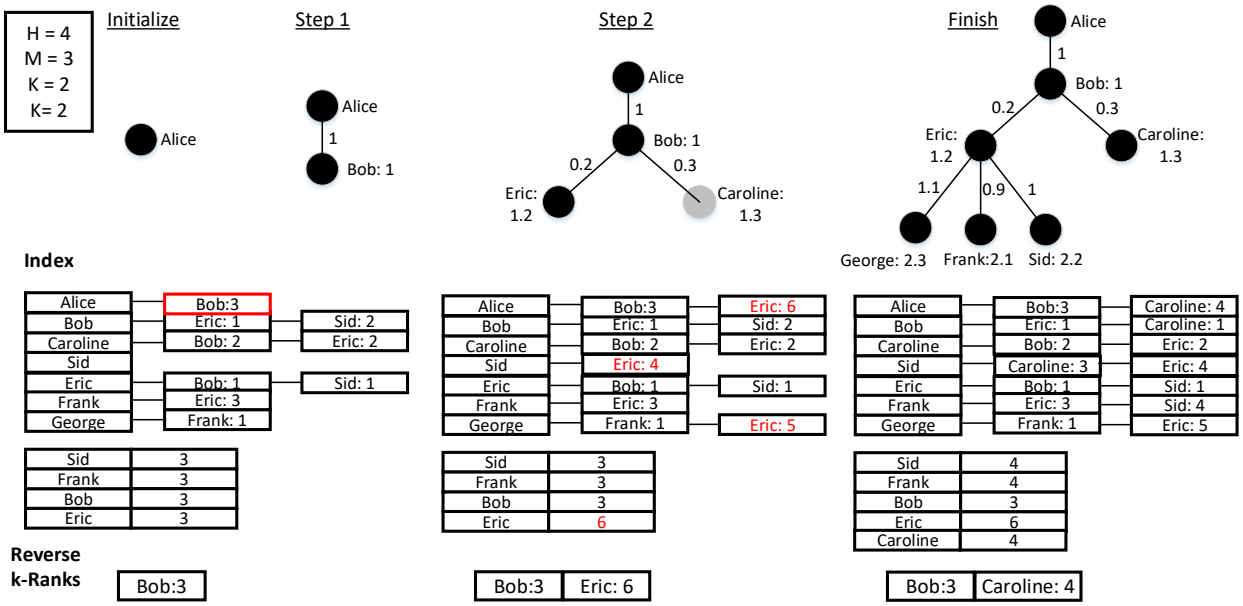


Figure 4: Index Update

There are 321,270 nodes and 800,172 edges in the network. Each store is mapped to the nearest network node. The weight on an edge (u, v) models the travel time between nodes u and v.

6.2 Effectiveness Analysis

To demonstrate the effectiveness of graph based reverse k-ranks queries, we conduct analysis at different levels of granularity that illustrate the problems of reverse top-k and top-k queries and the superiority of reverse k-ranks queries.

6.2.1 *Coarse-grained Analysis*

In our coarse-grained analysis, we investigate the results of top-k and reverse top-k queries on the DBLP dataset.

Reverse top-k query. Table 3 shows some statistics about the result sizes of reverse top-k queries on the DBLP dataset. k varies from 5 to 100. From the results, we can see that the size of result sets is not balanced. Reverse top-k queries return results of different cardinality for different query nodes. No matter what the value of k is, there always exists a large percentage of query nodes with empty reverse top-k result sets. When we increase the value of k from 5 to 100, the number of query nodes whose result set is empty decreases, but it remains in the same order of magnitude. At the same time, the largest result set size increases to 6,385, which is impractical to users.

Table 3: Reverse Top-k Result Set Size

k	5	10	20	50	100
largest set size	327	560	1,031	2,596	6,385
# of empty set	315,424	240,378	190,105	155,927	148,238
# of small set (≤ 5)	1,004,448	757,906	529,390	301,321	213,192
# of large set (≥ 100)	332	3,765	32,686	158,412	311,874

Top-k query. The problem of the top-k queries is that they are unilateral, i.e., the nodes that the query node ranks highest may not rank the query node high as well. To illustrate this problem, we investigate whether query nodes and the returned nodes have each other in their top-k results. We use $\mathbb{1}(i, j)$ to indicate whether nodes i and j agree with each other:

$$\mathbb{1}(i,j) = \begin{cases} 1, & \text{if } i \in \text{top}_k[j] \text{ and } j \in \text{top}_k[i] \\ 0, & \text{otherwise} \end{cases}$$

where $\text{top}_k[j]$ is the set of k-nearest nodes to j. Then the *agreement rate* can be calculated as:

$$\text{agreement rate} = \frac{\sum_i \sum_{j \in \text{top}_k[i]} \mathbb{1}(i,j)}{\sum_i |\text{top}_k[i]|}$$

Table 4 shows the agreement rate for various values of k. From the result, we can see that only less than half of the nodes in a top-k result also include the query node in their own top-k lists, i.e., a low agreement rate. Therefore top-k queries cannot be used as a substitute of reverse top-k and reverse k-ranks queries.

Table 4: Agreement Rate of Top-k Queries on DBLP

k Value	5	10	20	50	100
Agreement Rate(%)	48.53	44.65	41.10	37.88	35.65

6.2.2 *Fine-grained Case Study*

Wellcome and Parknshop are the two most popular supermarket chains in Hong Kong, having branches almost everywhere. We randomly choose a Wellcome and a Parknshop supermarket nearby on Google Maps, locate their nearby representative communities, and measure the road network distance between them as shown in Figure 5. In this case study, as we can see, the nearest representative community to Parknshop is B, however, if someone lives in B, he/she will prefer Wellcome to Parknshop since Wellcome is nearer. Instead, A and D would prefer Parknshop over Wellcome. Thus, the result of a top-1 query for Wellcome and Parknshop (B in both cases) is less meaningful for recommendation or advertisement compared to the result of the reverse-1 ranks query (B and A, respectively).

The reverse top-1 query here returns $\{A, D\}$ for Parknshop and $\{B, C, E, F, G\}$ for Wellcome, which is reasonable compared to

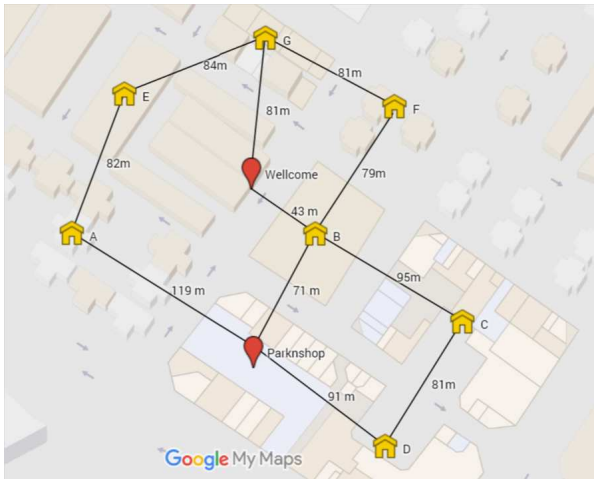


Figure 5: Case Study of Wellcome and Parknshop

Table 5: Parameters (default values in bold)

Parameter	Values
k	5,10,20,**50**,100
$h = H/\|V\|$	0.03,0.05,0.07,**0.1**,0.15
$m = M/\|V\|$	0.03,0.05,0.07,**0.1**,0.15
hub strategy	Random, **Degree First**, Closeness First

top-1 query, though with unfixed size. However, the relatively large size of results also means higher cost for promotion for the companies. On the other hand, the reverse k-ranks query defines an *ordering* of the communities with respect to their preferences on the supermarkets which can be used to prioritize market promotion to the communities that have higher chances to use it, in case of a limited budget.

6.3 Efficiency Analysis

In this section, we evaluate the performance of the reverse k-ranks approaches proposed in this paper, namely the static SDS-tree (Section 3), the dynamic SDS-tree (Section 4), and the dynamic SDS-tree with index (Section 5). We first measure the cost of the approaches as a function of different parameter values of the problem and the index. Then, we evaluate the effectiveness of the bounds used by the dynamic SDS-tree method. Next, we assess the cost of updating the index, and finally we study the efficiency of our methods on bichromatic instances of the problem.

6.3.1 Varying the Parameter Values

We evaluate the performance of our methods as a function of the following parameters: (1) size of the result set k, (2) percentage of hub nodes $h = H/|V|$, (3) percentage of ranked nodes in each index entry $m = M/|V|$, (4) hub selection strategy. Table 5 summarizes the range of values and the default value of each parameter. We measure performance by means of (i) query time and (ii) pruning power. For each setting of parameter values we run 1000 random queries and average the measures. Pruning power is measured by the average number of times the Rank Refinement function is called (we call this measure Rank Refinement in the following). The larger the Rank Refinement value is, the lower the pruning power of the method is.

Effect of k. To study the effect that the result size k has in the performance of reverse k-ranks queries, we fix the other parameters to their default values (see Table 5), and vary k from 5 to 100. As Figure 6 shows, the evaluation cost increases with k, which is consistent with the expectation that the search space and the number of candidates increases with k. Note that the dynamic approach has significantly reduced average query time for both datasets, which can be explained by the greatly reduced number of rank-refinements. The indexing method further reduces the query time to less than a few seconds, with the help of the precomputed index. We observe that the index has a greater effect on time for smaller values of k on DBLP, which can be explained by the fact that the information needed by the queries has higher chance to already be present in the index for smaller values of k. Besides, as we can see, the index works better for Epinions than for DBLP when k is large. This is because of larger average degrees in DBLP. In DBLP, even though the number of Rank Refinement calls is reduced significantly, the reduction of the average query time is not as high. This is due to the fact that the index mainly helps to avoid rank refinements that are cheap (they correspond to cases where the resulting rank is low). DBLP is a much larger and denser graph than Epinions and it is often the case that the reverse k-ranks of a query q are nodes that are quite far from q (i.e., they have low ranks). Therefore the rank refinements that are not avoided can be quite expensive, so the numerous cheap rank-refinements that are prevented due to the index have less profound effect to the query cost.

Table 6: Results with Different h on DBLP

Hub Percentage h	Index Size	Query Time (s)	Rank Refinement
0.03	1.2G	2.80015	166.702
0.05	1.2G	2.77694	151.608
0.07	1.2G	2.74801	139.514
0.1	1.2G	2.60599	124.591
0.15	1.2G	2.59796	124.591

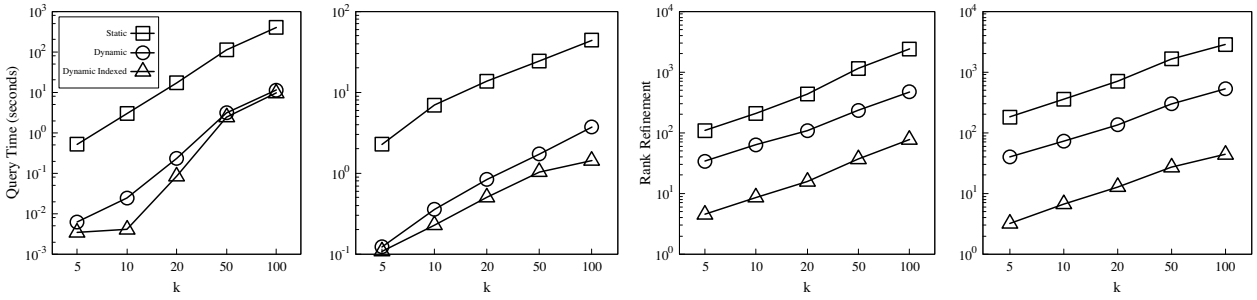
Table 7: Results with Different h on Epinions

Hub Percentage h	Index Size	Query Time (s)	Rank Refinement
0.03	25M	1.102102	70.431
0.05	27M	1.015720	66.483
0.07	29M	1.007760	63.699
0.1	30M	0.940826	59.044
0.15	32M	0.919234	51.488

In order to assess the effectiveness of our framework, we also ran tests using a naive reverse k-ranks method, described at the end of Section 2. Given the query node q, this method naively computes $Rank(p, q)$ for every node $p \in V$, by running SSSP from p until q is encountered. The top-k $Rank(p, q)$ values are tracked and eventually returned to the user. For $k = 1$, the average runtime of this naive approach on Epinions is 701.18s with 75878 Rank Refinements (For DBLP dataset, the average runtime of this naive approach is over 2000s, which is terminated by us manually), i.e., the naive method is significantly slower than the static SDS-tree approach and the dynamic SDS-tree approach with index.

Effect of hub percentage. The second parameter of which the effect we investigate is $h = H/|V|$, i.e., the percentage of hubs in the index. As shown in Tables 6 and 7, on both datasets, the average query time and the number of Rank Refinement calls decrease as h increases. Still, even for small h values, the query cost is not much higher compared to the default value. On the other hand, the index size is bounded and does not increase significantly as h increases.

Effect of index percentage. The third parameter that we study is $m = M/|V|$, i.e., the percentage of precomputed neighbors for



(a) DBLP Time (b) Epinions Time (c) DBLP Rank (d) Epinions Rank

Figure 6: Results with Different k

Table 8: Results with Different m on DBLP

Index Percentage m	Index Size	Query Time (s)	Rank Refinement
0.03	1.2G	2.756210	125.358
0.05	1.2G	2.723310	124.952
0.07	1.2G	2.626200	124.669
0.1	1.2G	2.605990	124.591
0.15	1.2G	2.577100	124.291

Table 9: Results with Different m on Epinions

Index Percentage m	Index Size	Query Time (s)	Rank Refinement
0.03	22M	0.970253	60.900
0.05	26M	0.963204	60.201
0.07	28M	0.954575	59.817
0.1	30M	0.940826	59.044
0.15	33M	0.912329	57.963

each node. As shown in Tables 8 and 9, similar to the effect of h, both the average query time and Rank Refinement calls decrease as m increases on the two real datasets. Again, the differences are not dramatic compared to the default value of m.

Effect of hub selection stategy. Next we test the effect of different hub selection strategies. As Table 10 shows, the Degree First and the Closeness First strategies are superior compared to the baseline Random strategy. Although on both DBLP and Epinions, Degree First is the winner, Closeness First performs quite similarly.

Table 10: Results with Different Hub Selection Strategies

Dataset		Random	Degree First	Closeness First
DBLP	Query Time (s)	2.861070	2.605990	2.665950
	Rank Refinement	169.500	124.591	135.132
Epinions	Query Time (s)	1.07950	0.940826	0.948437
	Rank Refinement	80.347	59.044	59.409

6.3.2 *Bound Analysis*

In this experiment, we test the effect of the three components of the Rank lower bound of Theorem 2. We ran 1000 random queries on Epinions. For each query and each candidate node, we count how many times each component wins as a maximum. The results are shown in Table 11. As we can see, in most cases, the rank of the parent offers the tight-most bound. In addition, although simple, height is a useful bound especially when the candidate nodes are close to the query node (i.e., if the nodes have large degrees, or when k is small). However, the effect of height declines when the value of k increases. On the other hand, the Count component is not very effective when k value is small, since only around 1% of the pruned cases are due to this bound. Besides, this bound cannot be applied for directed graphs, and it also has significant space requirements (i.e. $\mathcal{O}(|V|)$). The Count component is useful only for candidate nodes that are quite far from the query node (i.e., if the nodes have small degrees, or when k is large).

Table 11: Bound Analysis of Theorem 2

k	1	5	10	20	50	100
Height wins	87.74%	39.35%	27.48%	17.96%	9.50%	5.80%
Count wins	0.00%	0.44%	0.71%	1.07%	1.76%	2.38%
Parent wins	12.26%	60.21%	71.81%	80.97%	88.74%	91.82%

We also test the performance on Epinions by choosing 1000 queries with largest degree or fewest degree, using the dynamic SDS-tree algorithm with the four different bound strategies listed below:

- Dynamic-Parent: Rank(p,q)$\geq$ max(Rank(p',q))
- Dynamic-Count: Rank(p,q)$\geq$ max(Rank(p',q), $p.lcount$)
- Dynamic-Height: Rank(p,q)$\geq$ max(h, Rank(p',q))
- Dynamic-Three: Rank(p,q)$\geq$ max(h, Rank(p',q), $p.lcount$)[6]

The results are shown in Table 12 and Table 13. They also demonstrate that Height Component works better for nodes with large degrees while Count component works better for nodes with small degrees. As shown in Table 12, Height Component can significantly reduce Rank Refinement calls especially for small k values, while Count component brings in extra cost, which further increases query time when combining the three components compared to when using only the Height Component (an exception is the case of a large k value, i.e. k=100). On the other hand, the Count Component works better when the degree of the query node is low (i.e. Table 13) and the value of k is large.

In general, the rank of the parent offers the tight-most bound, while the Height Component helps when k is small or the query node's degree is large (i.e., candidate nodes are close to the query node). The Count Component is the least useful, being effective only when k is large or the query node has a small degree.

[6]Here we use the same symbol as in Section 4, where the depth of node p is h, node p' is the parent node of node p, and node p has been visited during the rank refinements of other nodes for $p.lcount$ times before the refinement of node p itself.

Table 12: Results with Different Bound Strategies Tested on Queries with Max Degree

k		1	5	10	20	50	100
Dynamic-Parent	Query Time (s)	0.001347	0.001348	0.001388	0.001586	0.003025	0.006705
	Rank Refinement	124.494	125.208	134.046	193.684	744.034	1738.360
Dynamic-Count	Query Time (s)	0.001385	0.001386	0.001419	0.001603	0.002872	0.006229
	Rank Refinement	124.211	124.913	133.425	189.876	690.931	1554.540
Dynamic-Height	Query Time (s)	0.000584	0.000618	0.000687	0.000856	0.001507	0.004156
	Rank Refinement	1.000	5.096	11.048	30.802	185.770	584.523
Dynamic-Three	Query Time (s)	0.000645	0.000680	0.000743	0.000917	0.001541	0.004076
	Rank Refinement	1.000	5.096	11.046	30.542	178.782	541.056

Table 13: Results with Different Bound Strategies Tested on Queries with Min Degree

k		1	5	10	20	50	100
Dynamic-Parent	Query Time (s)	0.001581	0.105760	0.258846	0.533142	1.388120	2.738640
	Rank Refinement	1.568	20.987	32.134	56.732	134.454	253.125
Dynamic-Count	Query Time (s)	0.001599	0.100026	0.269425	0.513132	1.318810	2.706620
	Rank Refinement	1.568	20.712	31.827	56.443	132.923	248.364
Dynamic-Height	Query Time (s)	0.001614	0.106397	0.271879	0.500304	1.358450	2.736410
	Rank Refinement	1.359	14.27	28.595	54.818	134.091	253.103
Dynamic-Three	Query Time (s)	0.001668	0.111674	0.257887	0.499597	1.274270	2.696320
	Rank Refinement	1.359	14.229	28.411	54.576	132.588	248.342

6.3.3 Index Update Analysis

In the next experiment, we evaluate the effectiveness of index updates. We randomly select 6,000 queries for each of the two real datasets and applied these queries in four different ways. We divided the 6,000 queries into n sets of the same size, i.e., for $n = 6, 3, 2, 1$, each set has 1,000, 2,000, 3,000, 6,000 queries respectively. Then, we run the Dynamic SDS-Tree with Index method n times and computed the average query time (including index update time) as well as the average number of Rank Refinement calls. All four different times apply the same 6,000 queries in the same order, the only difference being that when $n > 1$, the index is initialized (i.e., reset) multiple times. For example, when $n = 6$, the index is initialized and updated during the first 1,000 queries, then re-initialized, for the next 1,000 queries, etc. The objective is to understand whether and how the index performance improves as the index gets updated. Table 14 shows the average runtime and conducted rank-refinements per query. We can see that the more the index evolves the more rank refinements can be avoided and the more the average query time decreases.

Table 14: Results with Index Update

Dataset		Query Time (s)	Rank Refinement
DBLP	1,000	2.6287438	130.255
	2,000	2.530356	126.634
	3,000	2.486031	123.263
	6,000	2.228565	115.641
Epinions	1,000	1.179105	61.407
	2,000	0.985524	50.599
	3,000	0.924317	45.206
	6,000	0.544288	34.958

Table 15 shows the cost for initializing the index for various values of h and m. Although the times are quite high (especially for large values of h and m), this one-time cost pays off, because the index can help to achieve substantial cost savings for reverse k-ranks queries, as already shown.

Table 15: Index Construction Time (hours)

h	m	DBLP	Epinions
0.03	0.1	2.68	0.01
0.05	0.1	4.08	0.02
0.07	0.1	6.21	0.03
0.1	0.1	8.94	0.04
0.15	0.1	12.94	0.06
0.1	0.03	2.95	0.01
0.1	0.05	3.92	0.02
0.1	0.07	6.40	0.03
0.1	0.1	8.94	0.04
0.1	0.15	12.79	0.06

6.3.4 Bichromatic Queries

Although our solutions are only described in a monochromatic context, they are readily available for the case where the graph nodes are divided into two classes and the nodes are ranked with respect to where they are ranked by nodes that belong to the other class. Examples of bichromatic top-k, reverse top-k and reverse k-ranks queries have been shown at the case study of Section 6.2.2. In this section, we evaluate the performance of our methods for bichromatic reverse k-ranks queries. For the sake of completeness, we first provide a problem definition.

DEFINITION 3. *(Bichromatic $Rank(s,t)$) Consider a bichromatic weighted graph $G = (V, E)$, consisting of a set of nodes $V = V_1 \cup V_2$ and a set of edges E. Each edge in E carries a non-negative weight. For any two nodes $s \in V_1$, $t \in V_2$, let $d(s,t)$ denote the shortest path distance from s to t, which is defined by summing up the weights of the edges along the shortest path from s to t. Let $S \subset V_2$ be the set of nodes that satisfy $\forall p_i \in S, d(s,p_i) < d(s,t)$ and $\forall p_j \in (V_2 - S - \{t\}), d(s,p_j) \geq d(s,t)$. Then, $Rank(s,t) = |S| + 1$ where $S \subset V_2$ and $|S|$ is the cardinality of S.*

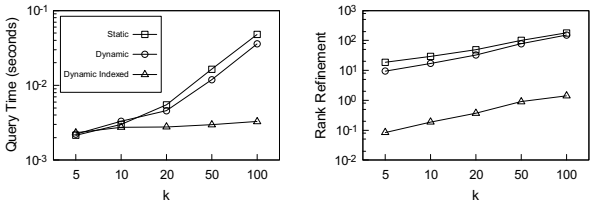
DEFINITION 4. *(Reverse k-Ranks Query on a Bichromatic Graph)*

Given a bichromatic weighted graph $G = (V, E)$, where $V = V_1 \cup V_2$, a query node $q \in V_2$ and a positive integer k, the reverse k-ranks query on a bichromatic graph returns a subset T of V_1, such that $|T| = k$ and $\forall p_i \in T$, $\forall p_j \in (V_1 - T)$, $Rank(p_i, q) \leq Rank(p_j, q)$.

In a reverse k-ranks query on a bichromatic graph, the query node is of one type, while the returned results are of another type. All our proposed methods can be used here with little modification: only for the nodes of the same type as the result set we need to do rank refinement, and only the nodes of the same type as the query node need to be counted during rank refinement. This is consistent with our case study with the communities and supermarkets; i.e., the management of a supermarket may use a reverse k-ranks query to find out which k communities the supermarket has higher chances to attract.

We use the SF road network (described in Section 6.1) to test the efficiency of our three proposed methods. We extract from the graph the nodes which are the nearest ones to 408 real stores and mark them as store nodes, while all other nodes are considered to be community nodes. We vary k from 5 to 100 in the experiment and measure the average query time and the average number of rank refinements per query, as performance metrics. The results are plotted in Figure 7. As we can see, when k is small, even though the Dynamic and Dynamic-Indexed methods can reduce the number of rank refinements, their average query time is not reduced; the cost of these methods is higher than the static approach for k=5. This is because the overhead cost by the data structures maintained by the Dynamic and Dynamic-Indexed methods. However, for larger k values, the superiority of the dynamic approaches and the index becomes apparent. Note that in this case of a sparse graph, the index approach is much more efficient compared to the static/dynamic SDS-tree method without index.

The general conclusions from our experimental study are as follows: (1) The reverse k-ranks query produces more useful results compared to the top-k query and the reverse top-k in recommendation applications, where a ranked set of objects of certain size needs to be recommended to the query object q; (2) Our filter-and-refinement framework is very efficient compared to the naive approach; (3) In the dynamic SDS-tree method, the parent-based and height-based bounds have higher applicability and effectiveness compared to the count-based bound, but count-based bound also has its applicable scenario; (4) The index is more effective for sparse graphs and for medium to high values of k.



(a) SF Time (b) SF Rank

Figure 7: Performance on a Bichromatic Network

7. RELATED WORK

Ranking queries have been widely used in many applications and have many variants. The most general one is the top-k query which returns the k objects with the highest scores based on a ranking function. Recently, reverse ranking queries emerged, which rank from the perspective of result objects, not the query objects.

Top-k query. Top-k queries have already been studied extensively for decades [10]. They return a list of objects which are ranked using an aggregate function that applies on their features. The most famous algorithms for top-k queries are Fagin's Threshold Algorithm (TA) and No Random Accesses (NRA) [5]. They are designed for rank-combining sorted lists of objects based on different attributes (features). TA allows both sequential and random accesses to these lists, whereas NRA allows only sequential accesses. Both algorithms use bounds for the top-k results, based on the information accessed so far and terminate as soon as the current top-k results are guaranteed to be the final ones, aiming at minimizing the accesses to the input lists. Mamoulis et al. [15] proposed an improved version of NRA, which is designed to minimize the number of object accesses, the computational cost, and the memory requirements of top-k search with monotone aggregate functions. If the ranking function is defined based on the distance of the objects to a pivot object, top-k queries are referred to as k nearest neighbor (k-NN) queries. k-NN queries have extensively been studied in spatial databases [9]. The single-source shortest path (SSSP) algorithm (a simple adaptation of Dijkstra's algorithm) can be used to evaluate k-NN queries on graphs.

Reverse k-NN query. The reverse k nearest neighbor (RKNN) query returns a set of query objects that have a given query point as one of their k nearest neighbors [24]. Preprocessing-based methods usually leverage index structures for efficient RKNN query evaluation. For example, the R-tree [8] is used for RKNN queries on spatial data [13, 20]. Yiu et al. [25] studied RKNN on large graphs using shortest path as the proximity measure. Similar to the reverse top-k query, which we review next, the result size of RKNN is not fixed, which may limit its application.

Reverse top-k query. The reverse top-k query returns the aggregate functions which rank a given query object highest. Vlachou et al. [21] presented an efficient threshold-based algorithm that eliminates candidates, without having to evaluate any top-k queries using the result functions. Furthermore, they introduced an indexing structure based on materialized reverse top-k views in order to speed up the computation of reverse top-k queries. These techniques were improved later in [22]. Ge et al. [6] proposed methods that compute all top-k queries in batch by applying the block indexed nested loops paradigm and a view-based algorithm. Yu et al. [26] studied reverse top-k queries when applied on large graphs, using Random Walk with Restart distance between nodes as the ranking function. Essentially, such reverse top-k queries on graphs are equivalent to RKNN on graphs, but using a different distance measure. As explained in [27], reverse top-k (and RKNN) queries only give results for query objects which are 'hot' (i.e., easily reachable by many other nodes), while most 'cold' objects get empty or too small result sets.

Reverse k-ranks query. To solve the aforementioned problem of RKNN queries, Zhang et al. [27] propose a new ranking query: the reverse k-ranks query in vector spaces. The reverse k-ranks query returns the k objects with the smallest Rank(w,q) values, where Rank(w,q) denotes the number of objects ranking higher than q for the same ranking function w. As opposed to reverse top-k and RKNN queries, the result set size of a reverse k-ranks query is fixed.

8. CONCLUSION

This paper is the first-time ever study of reverse k-ranks queries

over large graphs. We have shown through real-life case studies that reverse top-k queries may produce unsatisfactory results; therefore, there is a need for the efficient support of reverse k-ranks queries. Then, we proposed a filter-and-refinement framework for evaluating reverse k-ranks queries, based on the construction of a SDS-tree and the dynamic refinement of its nodes. We also proposed an indexing technique that can further improve the performance of the framework. Our experimental evaluation which uses three real large-scale graphs of different characteristics confirms the efficiency of the proposed techniques. In the future, we plan to study reverse k-ranks queries for other node similarity measures (i.e. PageRank, Personalized PageRank and SimRank), which require radically different approaches.

Acknowledgements

This work was supported by grants GRF 17205015 and 17201414 from Hong Kong RGC.

9. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360. ACM, 2013.

[2] A. Bavelas. Communication patterns in task-oriented groups. *Journal of the acoustical society of America*, 22:725–730, 1950.

[3] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

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

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[6] S. Ge, L. H. U, N. Mamoulis, and D. W. Cheung. Efficient all top-k computation - a unified solution for all top-k, reverse top-k and top-m influential queries. *TKDE*, 25(5):1015–1027, 2013.

[7] Z. Guan, J. Bu, Q. Mei, C. Chen, and C. Wang. Personalized tag recommendation using graph-based ranking on multi-type interrelated objects. In *SIGIR*, pages 540–547. ACM, 2009.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM Press, 1984.

[9] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[10] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[11] M. Jiang, A. W. Fu, and R. C. Wong. Exact top-k nearest keyword search in large networks. In *SIGMOD*, pages 393–404. ACM, 2015.

[12] KONECT. DBLP network dataset. http://konect.uni-koblenz.de/networks/dblp_coauthor, 2015.

[13] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *SIGMOD*, pages 201–212. ACM, 2000.

[14] S. Lo and C. Lin. WMR–A graph-based algorithm for friend recommendation. In *WI*, pages 121–128. IEEE Computer Society, 2006.

[15] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-*k* aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3):19, 2007.

[16] F. Meng, D. Gao, W. Li, X. Sun, and Y. Hou. A unified graph model for personalized query-oriented reference paper recommendation. In *CIKM*, pages 1509–1512. ACM, 2013.

[17] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.

[18] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[19] P. Symeonidis, E. Tiakas, and Y. Manolopoulos. Product recommendation and rating prediction based on multi-modal social networks. In *RecSys*, pages 61–68. ACM, 2011.

[20] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *PVLDB*, pages 744–755. Morgan Kaufmann, 2004.

[21] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, pages 365–376. IEEE Computer Society, 2010.

[22] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *SIGMOD*, pages 481–492. ACM, 2013.

[23] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815. IEEE Computer Society, 2011.

[24] S. Yang, M. A. Cheema, X. Lin, and W. Wang. Reverse k nearest neighbors query processing: Experiments and analysis. *PVLDB*, 8(5):605–616, 2015.

[25] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *TKDE*, 18(4):540–553, 2006.

[26] A. W. Yu, N. Mamoulis, and H. Su. Reverse top-k search using random walk with restart. *PVLDB*, 7(5):401–412, 2014.

[27] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *PVLDB*, 7(10):785–796, 2014.