Finding the Optimal Path over Multi-Cost Graphs

Yajun Yang [†1], Jeffrey Xu Yu [‡2], Hong Gao [†3], Jianzhong Li [†4]
[†]Harbin Institute of Technology, China
[‡]The Chinese University of Hong Kong, China
[1,3,4]{yjyang, honggao, lijzh}@hit.edu.cn, [2] yu@se.cuhk.edu.hk

ABSTRACT

Shortest path query is an important problem in graphs and has been well-studied. However, most approaches for shortest path query are based on single-cost (weight) graphs. In this paper, we introduce the definition of multi-cost graph and study a novel query: the optimal path query over multi-cost graphs. We propose a *best-first* branch and bound search algorithm with two optimizing strategies. Furthermore, we propose a novel index named k-cluster index to make our method more space and time efficient for large graphs. We discuss how to construct and utilize k-cluster index. We confirm the effectiveness and efficiency of our algorithms using real-life datasets in experiments.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms

Keywords

Multi-cost graphs, optimal path, non-linear functions

1. INTRODUCTION

Graphs have been widely used to model complex relationships among various entities in real applications. Shortest path query in graphs is an important problem and has been well-studied. However, most of existing works assume that there is a single cost type on edges. In fact, the edges may have multiple cost types to describe the relationships among various entities. In a transportation network, there are several cost types to measure an edge(representing a highway) between two cities, such as length, traveling time, toll fee, etc. We call a graph *multi-cost graph* if there are several cost types on edges. In real applications, these cost types coexist and may collectively affect the decisions of users. It is unadvisable to choose a shortest path by a single cost type. For example, in transportation network, the summed toll fee of the path with the shortest length may be too expensive to accept. In this case, users prefer to choose a path which is slightly longer than the shortest path but its toll fee is very low. Therefore, it is important to find an optimal path under global consideration according to user's preference.

We utilize a score function $f(\cdot)$ to measure the importance of paths in multi-cost graphs. $f(\cdot)$ calculates an overall score for a path according to all cost types of this path. Given a score function $f(\cdot)$, starting vertex s and ending vertex t, in this paper, our objective is to find a path from s to t that has the minimum score. This path is said to be the *optimal path* from s to t under function $f(\cdot)$. To the best of our knowledge, our paper is the first research work about the optimal path query over multi-cost graphs.

All existing works for shortest path problem utilize the following property: any sub-path of a shortest path is also a shortest path. Unfortunately, this property does not hold in multi-cost graphs if score function is non-linear(detailed in section 2). Thus, all existing methods cannot solve the optimal path problem over multi-cost graphs. As analysis in the works about transportation problem[4], the non-linear score functions are existent and reasonable.

The main contributions are summarized below. Given a multi-cost graph G, score function $f(\cdot)$, starting vertex s and ending vertex t. First, we define a novel optimal path query over multi-cost graphs. Second, we propose a *best-first* branch and bound search algorithm with two optimizing strategies. Third, we propose a novel index for multi-cost graphs, named k-cluster index, which makes our method more efficient for large graphs. k-cluster index is with lower space cost than naive index. Fourth, we introduce how to answer the optimal path query over multi-cost graphs by k-cluster index. Finally, we confirm the effectiveness and efficiency of our algorithms using real-life datasets.

2. PROBLEM STATEMENT

A multi-cost graph is a simple directed graph, denoted as $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge $e \in E$ is represented by $e = (u, v)$, $u, v \in V$, e is called u's outgoing edge or v's incoming edge and v(or u) is called u(or v)'s outgoing(or incoming) neighbor. Each edge $e \in E$ is assigned a cost vector $cost(e)$, $cost(e) = (c_1, c_2, \cdots, c_d)$, where c_i is the i-th cost value of edge e according to the d cost types involved in decision making. For example, in transportation network, an edge e between city A and B represents a highway from A to B. $cost(e) = (c_1, c_2, c_3)$ is a 3-dimensional cost vector of e, where c_1 would be the Euclidean distance between A and B, c_2 would be the driving time from A to B, and c_3 could be the toll fee, etc. In this paper, we assume $c_i \geq 0$. This assumption is reasonable, because the cost cannot be less than zero in real applications. Our work can be easily extended to handle undirected graphs, an undirected edge $e = (u, v)$ is equivalent to two directed edges $e_1 = (u, v)$

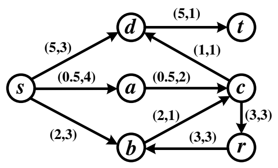

Figure 1: An example of multi-cost graph $G(V,E)$

and $e_2 = (v,u)$, where $cost(e_1) = cost(e_2) = cost(e)$. For simplicity, we only discuss directed graphs in following.

A path p is a sequence of vertices $(v_0, v_1, \cdots, v_l)$, where $v_i \in V(0 \le i \le l)$ and $e_i = (v_{i-1}, v_i) \in E(0 < i \le l)$. p is simple if and only if there is no repeated vertex in p, i.e., $v_i \neq v_j$, for any $i \neq j, 0 \le i, j \le l$. The cost vector $cost(p)$ of path p is the vector sum of its constituent edges. Let $\{e_1, e_2, \cdots, e_l\}$ be the set of constituent edges of path p and let $cost(p) = (c_1(p), c_2(p), \cdots, c_d(p))$ be the cost vector of path p. Then, $cost(p) = \sum_{j=1}^{l} cost(e_j)$. Here, $c_i(p) = \sum_{j=1}^{l} c_i(e_j)$, $0 \le i \le d$. Note that $c_i(p)$ and $c_i(e_j)$ are the i-th cost value of $cost(p)$ and $cost(e_j)$ respectively.

Score function $f(\cdot)$ is an aggregate function specified by user in multi-dimensional space. For each data object, $f(\cdot)$ aggregates its values on all dimensions to one overall score. Generally, the best object is the object with the minimum score. In this paper, $f(\cdot)$ computes score of a path p according to $cost(p)$, i.e., $f(p) = f(c_1(p), c_2(p), \cdots, c_d(p))$. Here, $f(p)$ equals to $f(cost(p))$. For simplicity, we use $f(p)$ in following paper. We assume function $f(\cdot)$ is monotone increasing, i.e., for any two different paths p and p', if $(\forall i, c_i(p) \le c_i(p')) \wedge (\exists i, c_i(p) < c_i(p'))$, then $f(p) < f(p')$. The restriction of monotonicity is a common property and it is reasonable[2]. Its intuitive meaning is that: if all costs of a path p are less than that of another path p', then the overall score of p is at least good as p'. The definition of the optimal path over multi-cost graphs is given as below:

Definition 2.1: (**Optimal Path**) Given a multi-cost graph $G(V,E)$ and score function $f(\cdot)$, $s, t \in V$ are any two different vertices in G. Let $P_{s,t}$ represent the set of all simple paths from s to t in G. The optimal path from s to t, denoted as $sp(s,t)$, is a path in G that has the minimum score among all paths in $P_{s,t}$, that is, $f(sp(s,t)) \le f(p)$ for any $p \in P_{s,t}$. □

The problem of the optimal path query over multi-cost graphs is given as follows:

Problem Statement: Given a multi-cost graph $G(V,E)$, score function $f(\cdot)$, starting vertex s and ending vertex t. Find the optimal path $sp(s,t)$ from s to t such that $f(sp(s,t))$ is minimum.

If score function $f(\cdot)$ is linear, i.e., for any two edges e_i and e_j $(i \neq j)$, $f(e_i + e_j) = f(e_i) + f(e_j)$, We only need to consider $f(e)$ as edge e's single-one weight. We apply existing algorithm, e.g., Dijkstra algorithm, to compute the shortest path according to weight $f(e)$. This shortest path is exactly the optimal path for our problem. Otherwise, there is another path p' such that $f(p') < f(p)$. By the linearity of score function $f(\cdot)$, we have $f(p') = f(\sum_{i=1}^{l} e'_i) = \sum_{i=1}^{l} f(e'_i) < f(p) = f(\sum_{i=1}^{r} e_i) = \sum_{i=1}^{r} f(e_i)$, which is in contradiction to the correctness of the shortest path (Dijkstra) algorithm.

If score function $f(\cdot)$ is non-linear, i.e., $f(e_i + e_j) \neq f(e_i) + f(e_j)$, then existing methods cannot solve our problem. The frameworks of these methods are that: build an index to maintain the shortest paths between any two vertices in index. Given a query, algorithms first retrieve the shortest paths to be visited inside index and then concatenate them by the shortest paths outside index. All these methods utilize the following property: any sub-path of a shortest path is also a shortest path. Hence, they only need to maintain the shortest paths for any two vertices in index. However, the optimal sub-path property does not hold in multi-cost graphs if score function is non-linear. As shown in Fig. 1, the score function is $f(x,y) = x^2 + y^2$. We find that the optimal path from s to r is $s \to a \to c \to r$. Here, the sub-path $p : s \to a \to c$ is not the optimal path from s to c, because its score is $f(1,6) = 37$, which is larger than the score $f(4,4) = 32$ of path $p' : s \to b \to c$. It states the sub-path of the optimal path may not be an optimal path.

3. BRANCH AND BOUND ALGORITHM

In this section, we propose a *best-first* branch and bound search algorithm with two optimizing strategies.

3.1 Basic Algorithm

Given a multi-cost graph $G(V,E)$, score function $f(\cdot)$, starting vertex s and ending vertex t, all possible paths started from s in G can be organized in a search tree. Here, the root node represents starting vertex set $\{s\}$, and any non-root node represents a path started from s. Let C and C' be two nodes in the search tree and they represent two different paths started from s in G. The node C is the parent of another node C' if they satisfy the following two conditions: (i) $C \subset C'$ and $|C'| = |C| + 1$; (ii) the only vertex $v \in C' \setminus C$ satisfies $v \notin C$ and $v \in N^+(u)$, where u is the ending vertex of C and $N^+(u)$ is the outgoing neighbor set of u. $C \subset C'$ implies C is a path prefix of C', $|C|$ represents the number of vertices in C. $v \notin C$ guarantees there does not exist circle if append v to C, i.e., C' is a simple path. With the search tree, the problem to find the optimal path from s to t in G becomes a tree searching problem. That is to find a node C, where the ending vertex of C is t, such that in the search tree C has the minimum score $f(C)$ no larger than any other C' whose ending vertex is t. In the following, we use C to refer a node in the search tree as well as the path it representing.

We use a min-heap H to maintain the nodes to be visited in the search tree. The nodes in H are sorted by their scores. We initialize H only with the starting vertex set $\{s\}$. Our algorithm performs *best-first* branch and bound search over the search tree by repeatedly popping up the top element C, which has the minimum score $f(C)$ in H. Below, let τ indicate the current minimum $f(C)$ of C whose ending vertex is t. Initially, $\tau = \infty$.

For a node C, if $f(C) > \tau$, then there does not exist the optimal path from s to t in the subtree rooted at C. Thus, we can prune this branch safely. Lemma 3.1 guarantees the correctness of this pruning rule.

Lemma 3.1: *For any two nodes C and C' in the search tree, if C is an ancestor of C', then $f(C') > f(C)$.* □

The correctness of Lemma 3.1 can be proved by the monotonicity of function $f(\cdot)$.

When a node C pops up from H, algorithm will expand C by processing the children of C in search tree. Assume that the ending vertex of C is u. For any $v \in N^+(u)$, we first check whether there exists a circle if append v to C, i.e., $v \in C$ or not. If $v \notin C$, we calculate the score $f(C')$ for node $C' = C \cup \{v\}$, C' is a child of C and its ending vertex is v. In case of $v \neq t$, if $f(C') \ge \tau$, by Lemma 3.1, we can safely prune the subtree rooted at C'. If $f(C') < \tau$, we insert C' into H. In case of $v = t$, C' is a path from s to t. If $f(C') < \tau$, we use C' instead of the current optimal path from s to t and update τ by $f(C')$. C' will not be inserted into H when its ending vertex is t.

The algorithm terminates when $H = \emptyset$, or the score $f(C) \ge \tau$ for the top element C in H. When the algorithm terminates, the

Algorithm 1 FIND-$sp(s,t)$-*best-first*-SEARCH $(G, s, t, f(\cdot))$

Input: G, starting vertex s, ending vertex t and function $f(\cdot)$
Output: optimal path $sp(s,t)$ based on function $f(\cdot)$.

```
1: $\tau \leftarrow \infty$, $sp(s,t) \leftarrow \emptyset$, $H \leftarrow \{s\}$, $cost(\{s\}) \leftarrow 0$;
2: while $\mathcal{H} \neq \emptyset$ do
3:     let $C$ be the path by popping up the top element from $\mathcal{H}$ and $end(C) = u$; //$end(C)$ is the ending vertex of $C$.
4:     if $f(C) \geq \tau$ then
5:         break;
6:     for each vertex $v$, $v \in N^+(u) \wedge v \notin C$ do
7:         $C' \leftarrow C \cup \{v\}$; $f(C') \leftarrow f(C + (u,v))$;
8:         if $v = t$ then
9:             if $f(C') < \tau$ then
10:                $\tau \leftarrow f(C')$; $sp(s,t) \leftarrow C'$; continue;
11:        else
12:            $LB(C') \leftarrow f(C' + \Phi_{v,t})$;
13:            if $f(C') \geq \tau$ then
14:                prune the subtree rooted at $C'$; continue;
15:            if $\exists p \in \mathsf{SKYP}(s,v)$, $p \prec C'$ then
16:                prune the subtree rooted at $C'$; continue;
17:            else
18:                $\mathsf{SKYP}(s,v) \leftarrow \mathsf{SKYP}(s,v) \cup \{C'\}$;
19:                for each $p \in \mathsf{SKYP}(s,v)$ do
20:                    if $C' \prec p$ then
21:                        $\mathsf{SKYP}(s,v) \leftarrow \mathsf{SKYP}(s,v) - \{p\}$;
22:            if $LB(C') \geq \tau$ then
23:                prune the subtree rooted at $C'$; continue;
24:            insert $C'$ into $H$ according to $f(C')$;
25: return $sp(s,t)$, $\tau$;
```

current optimal path corresponding to τ is the answer. The *best-first* branch-and-bound algorithm is shown in Algorithm 1.

3.2 Pruning Rules

To further enhance the power of pruning, we develop two pruning rules. We first introduce the definition of skyline path before giving these rules.

Definition 3.1: (**Path Dominate**) Given a multi-cost graph $G(V,E)$, p and p' are two different paths in G. We say p dominates p', denoted as $p \prec p'$, iff for $\forall i (1 \leq i \leq d)$, $c_i(p) \leq c_i(p')$, and $\exists i (1 \leq i \leq d)$, $c_i(p) < c_i(p')$. Here, $c_i(p)$ and $c_i(p')$ are the i-th cost value of $cost(p)$ and $cost(p')$. □

Definition 3.2: (**Skyline Path**) Given a multi-cost graph $G(V,E)$ and two vertices $u, v \in V$. Let $P_{u,v}$ denote the set of all paths from u to v in G. A path p is said to be a skyline path from u to v if and only if p cannot be dominated by any other path p' in $P_{u,v}$, i.e., $\nexists p' \in P_{u,v}, p' \prec p$. □

Skyline path based pruning: For any vertex $u \in G$, we use a set $\mathsf{SKYP}(s,u)$ to maintain the skyline paths from s to u that have been searched up to now. Initially, $\mathsf{SKYP}(s,u) = \emptyset$. Given a node C, assume that the ending vertex of C is u. If $\exists p \in \mathsf{SKYP}(s,u)$, $p \prec C$, then the subtree rooted at C can be pruned safely, otherwise path C should be inserted into $\mathsf{SKYP}(s,u)$. In addition, if $\exists p \in \mathsf{SKYP}(s,u)$, $C \prec p$, then p should be removed from $\mathsf{SKYP}(s,u)$. Lemma 3.2 guarantees the correctness of this pruning rule.

Lemma 3.2: *Let C and C' are two different nodes in the search tree and both of their ending vertices are u. If $C' \prec C$, then the optimal path from s to t cannot be in the subtree rooted at C.* □

The correctness of Lemma 3.2 can be proved by the monotonicity of function $f(\cdot)$.

Next, we introduce the definition of the ***L**ower **B**ound of **O**ptimal **P**ath*(LBOP) and then give the second pruning rule.

Definition 3.3: (**Lower Bound of Optimal Path**(LBOP)) Given a multi-cost graph $G(V,E)$. Each edge $e \in E$ has a cost vector $cost(e)$, $cost(e) = (c_1(e), \cdots, c_d(e))$. $\mathcal{G}_1, \cdots, \mathcal{G}_d$ are d weighted graphs, $\mathcal{G}_i = (V,E)$. The weight of any edge e in $\mathcal{G}_i$ is $c_i(e)$. $\mathcal{G}_i$ is said to be a weighted graph based on i-th cost value in G. For any two vertices u and v, $\mathcal{P}_{u,v} = \{\mathcal{P}_{(u,v);1}, \cdots, \mathcal{P}_{(u,v);d}\}$ is called **the set of single-one cost shortest paths** from u to v, where $\mathcal{P}_{(u,v);i}$ is the weighted shortest path from u to v in $\mathcal{G}_i$. The cost of $\mathcal{P}_{(u,v);i}$ is $\phi_{(u,v);i}$. We say cost vector $\Phi_{u,v} = (\phi_{(u,v);1}, \cdots, \phi_{(u,v);d})$ is the **lower bound of optimal path** (LBOP) from u to v in G. □

LBOP **based pruning:** We pre-compute $\Phi_{u,v}$ for any two vertices u and v in G. Given a node C, let the ending vertex of C be u. We estimate a lower bound $LB(C)$ according to $\Phi_{u,t}$, $LB(C) = f(C + \Phi_{u,t})$. $LB(C)$ indicates the lower bound of the score of any path whose ending vertex is t in the subtree rooted at C. If $LB(C) \geq \tau$, then the subtree rooted at C can be pruned safely. Lemma 3.3 guarantees the correctness of this pruning strategy.

Lemma 3.3: *Let C be a node in search tree. $LB(C) \geq \tau$. If $\widetilde{C}$ is a path ended at t in the subtree rooted at C. then $f(\widetilde{C}) \geq \tau$.* □

The correctness of Lemma 3.3 can be proved by the monotonicity of function $f(\cdot)$.

4. K-CLUSTER INDEX

We note that the branch and bound algorithm need to maintain $\Phi_{u,v}$ for every two vertices u and v in G, which is too expensive for large graphs. In addition, the time cost is also expensive for large graphs. Next, we propose a new index named k-cluster index, which has small space cost and perform well on large graphs. We introduce what is k-cluster index and how to construct it.

4.1 What Is The K-Cluster Index?

Definition 4.1: (**K-Cluster**) Given a graph $G(V,E)$, k-cluster is a partition $\{V_1, \cdots, V_k\}$ of V, such that: (1) for $\forall V_i, V_j (i \neq j)$, $V_i \cap V_j = \emptyset$; (2)$V = \bigcup_{1 \leq i \leq k} V_i$. Each $V_i \subseteq V$ is called a cluster in G. A vertex v is said to be an *entry* of cluster V_i, if (1) $v \in V_i$; and (2) $\exists u$, $u \notin V_i \wedge u \in N^-(v)$. Similarly, A vertex v is said to be an *exit* of cluster V_i, if (1) $v \in V_i$; and (2) $\exists u$, $u \notin V_i \wedge u \in N^+(v)$. $N^-(v)$ and $N^+(v)$ are v's incoming and outgoing neighbor set, respectively. Entries and exits are also said to be the *border vertices*. □

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

A k-cluster index includes two parts: *inter-index* and *inner-index*.

Inter-index: Inter-index maintains the LBOP for every pair of border vertex and entry in G. It is essentially a matrix whose size is $(|V.exit| + |V.entry|) \times |V.entry|$. Each row represents a border vertex (entry or exit) u in G and each column represents an entry v in G. Each cell $A_{u,v}$ indicates the LBOP $\Phi_{u,v}$ from u to v.

Inner-index: Inner-index contains k sub-indices, where each sub-index I_x corresponds to a cluster V_x. Each sub-index I_x contains two components: (i) *Skyline-Path-Inner-Index* I_x^S; and (ii) LBOP-*Inner-Index* I_x^L.

Skyline-Path-Inner-Index I_x^S in cluster V_x is a collection of the sets of skyline paths for every pair of entry and exit in V_x, i.e., $I_x^S = \{\mathsf{SKYP}_x(u,v) | u \in V_x.entry, v \in V_x.exit\}$. $\mathsf{SKYP}_x(u,v)$ is the set of skyline paths from u to v in G_x, where G_x is the induced subgraph of V_x on G. Note that the paths in $\mathsf{SKYP}_x(u,v)$ only pass through the vertices in V_x.

LBOP-Inner-Index I^L_x in cluster V_x is a $|V_x| \times |V_x|$ matrix to maintain LBOPs for every two vertices $u, v \in V_x$. Similar to inter-index, each cell $A_{u,v}$ indicates the LBOP $\Phi_{u,v}$ from u to v.

By inter-index and LBOP-inner-index, we can compute $\Phi_{s,t}$ for any two vertices s and t in G. Given two vertices s and t, we first identify the clusters which contain s and t respectively. Let V_s and V_t denote the clusters that contain s and t respectively. If $V_s = V_t$, we can directly retrieve $\Phi_{s,t}$ from LBOP-inner-index I^L_s. If $V_s \neq V_t$, we give Lemma 4.1 to help us to compute $\Phi_{s,t}$.

Lemma 4.1: *Given a multi-cost graph G and two vertices s and t, let V_s and V_t be the clusters that contain s and t respectively, $V_s \neq V_t$. $\Phi_{s,t}$ is the* LBOP *from s to t, $\Phi_{s,t} = (\phi_{(s,t);1}, \cdots, \phi_{(s,t);d})$. For any entry $v \in V_t.entry$, $\Phi_{s,v}$ and $\Phi_{v,t}$ are* LBOP *from s to v and* LBOP *from v to t respectively. $\Phi_{s,v} = (\phi_{(s,v);1}, \cdots, \phi_{(s,v);d})$ and $\Phi_{v,t} = (\phi_{(v,t);1}, \cdots, \phi_{(v,t);d})$. Then, for $\forall i (1 \leq i \leq d)$, we have $\phi_{(s,t);i} = \min\{\phi_{(s,v);i} + \phi_{(v,t);i} | v \in V_t.entry\}$.* □

Proof Sketch: By the definition of LBOP, $\phi_{(s,t);i}$ is the cost of single-one cost shortest path $\mathcal{P}_{(s,t);i}$ in $\mathcal{G}_i$. Obviously, $\mathcal{P}_{(s,t);i}$ passes through a vertex in $V_t.entry$. Without loss of generality, assume this vertex is v. $\mathcal{P}_{(s,t);i}$ can be divided into two segments: (i) sub-path from s to v; and (ii) sub-path from v to t. $\phi_{(s,v);i}$ and $\phi_{(v,t);i}$ are the costs of shortest paths from s to v and from v to t respectively in $\mathcal{G}_i$, then $\phi_{(s,v);i} + \phi_{(v,t);i} \leq \phi_{(s,t);i}$. On the other hand, $\phi_{(s,t);i}$ is the minimum cost among all paths from s to t, then $\phi_{(s,t);i} \leq \phi_{(s,v);i} + \phi_{(v,t);i}$. Then we have $\phi_{(s,t);i} = \phi_{(s,v);i} + \phi_{(v,t);i}$. Next, we prove v is the vertex that minimizes $\phi_{(s,v);i} + \phi_{(v,t);i}$. This is correct otherwise $\mathcal{P}_{(s,t);i}$ is not the single-one cost shortest path in $\mathcal{G}_i$. Therefore, Lemma 4.1 is proved. □

We compute $\Phi_{s,t}$ in two cases: (i) $s \in V_s.entry \cup V_s.exit$; and (ii) $s \notin V_s.entry \cup V_s.exit$. For case (i), we first compute $\phi_{(s,v);i} + \phi_{(v,t);i}$ for all $v \in V_t.entry$. Note that $\phi_{(s,v);i}$ and $\phi_{(v,t);i}$ can be retrieved from inter-index and LBOP-inner-index I^L_t respectively. By Lemma 4.1, we select the minimum $\phi_{(s,v);i} + \phi_{(v,t);i}$ as $\phi_{(s,t),i}$, which is the i-th cost value of $\Phi_{s,t}$. For case (ii), $\phi_{(s,v);i}$ cannot be retrieved from inter-index directly. We compute $\phi_{(s,v);i}$ as follows: We first retrieve $\phi_{(s,u);i}$ from LBOP-inner-index I^L_s and $\phi_{(u,v);i}$ from inter-index for every $u \in V_s.exit$. By Lemma 4.1, we select $\min\{\phi_{(s,u);i} + \phi_{(u,v);i} | u \in V_s.exit\}$ as $\phi_{(s,v);i}$, and then compute $\phi_{(s,t),i}$ as similar as case (i).

4.2 How to Construct K-Cluster Index?

4.2.1 *Inter-index and* LBOP*-inner-index*

Inter-index and LBOP-inner-index I^L_x in each cluster V_x can be constructed easily. For LBOP-inner-index I^L_x, we adopt existing shortest path algorithm to compute $\Phi_{u,v}$ for any two vertex $u, v \in V_x$. For inter-index, we compute $\Phi_{u,v}$ for any $u \in V.entry \cup V.exit$ and $v \in V.entry$. Note that: if entry u and exit v are in the same cluster V_x, we do not need to maintain $\Phi_{u,v}$ in inter-index because it has been maintained in LBOP-inner-index I^L_x.

4.2.2 *Skyline-path-inner-index*

To construct I^S_x, we need to compute $\mathsf{SKYP}_x(u, v)$ for every pair of entry u and exit v in each cluster V_x. We propose a breadth-first branch and bound search algorithm, which is in a similar manner as Algorithm 1. We build a search tree rooted at $\{u\}$ for G_x like that in Algorithm 1 and use a queue Q to maintain the nodes to be searched, where G_x is the induced subgraph of V_x on G. When a node C pops up from queue, we expand C by processing the children of C. For a child C' of C, assume the ending vertex of C' is w. In case of $w \neq v$, if $\exists p \in \mathsf{SKYP}_x(u, v)$, $p \prec C'$, then we can prune the subtree rooted at C'. Otherwise, C' is inserted into Q. In case of $w = v$, if $\nexists p \in \mathsf{SKYP}_x(u, v)$, $p \prec C'$, then we insert p into $\mathsf{SKYP}_x(u, v)$. On the other hand, if $\exists p \in \mathsf{SKYP}_x(u, v)$, $C' \prec p$, we remove p from $\mathsf{SKYP}_x(u, v)$.

We also propose two pruning rules to improve efficiency.

Skyline path based pruning: We maintain a set $\mathsf{SKYP}_x(u, w)$ for each $w \in V_x$ in the searching process. For a node C whose ending vertex is w, if $\exists p \in \mathsf{SKYP}_x(u, w)$, $p \prec C$, then the subtree rooted at C can be pruned safely. Otherwise, we insert C into $\mathsf{SKYP}_x(u, w)$. In addition, if $\exists p \in \mathsf{SKYP}_x(u, w)$, $C \prec p$, we remove p from $\mathsf{SKYP}_x(u, w)$.

LBOP **base pruning**: For a node C whose ending vertex is w, consider $\Phi_{w,v}$ from w to v. We estimate a lower bound $LB(C)$ for C, $LB(C) = cost(C) + \Phi_{w,v}$. If $\exists p \in \mathsf{SKYP}_x(u, v)$, $p \prec LB(C)$, then the subtree rooted at C can be pruned safely.

The correctness of above two pruning rules can be proved as similar as that of pruning rules in Algorithm 1.

4.3 How to Partition Graph to K Clusters

There are several ways to partition a graph to k clusters. For different partitions, the number of entries and exits are different. In our problem, the less number of entries and exits makes the size of k-cluster index smaller. Intuitively, the less edges among different clusters results in the less number of entries and exits in graph. Thus, k-cluster partition problem is to find a partition such that the edges among k different clusters are sparse and the edges in a cluster are dense. It is a graph partition problem and this problem has been well studied. We adopt the multi-level graph partitioning technique proposed by Karypis et al. in [1], which is an efficient partition algorithm.

5. QUERY PROCESSING

Given an optimal path query from s to t, we construct a shrunk graph $\bar{G} = (\bar{V}, \bar{E})$. $\bar{V}$ includes three parts: (i) V_s; (ii) V_t; and (iii) $\bigcup_{x \neq s,t}(V_x.entry \cup V_x.exit)$. V_s and V_t are the clusters that contain s and t respectively. $\bar{E}$ also includes three parts: (i) $(u, v) \in \bar{E}$, iff $((u, v) \in E) \wedge ((u, v \in V_s) \vee (u, v \in V_t))$; (ii) $(u, v) \in \bar{E}$, iff $((u, v) \in E) \wedge ((u \in V_x.exit) \wedge (v \in V_y.entry))$, where $V_x \neq V_y$; and (iii) we create m new edges $\{(u, v)^1, \cdots, (u, v)^m\}$ for any entry $u \in V_x.entry$ and any exit $v \in V_x.exit$, where $V_x \neq V_s$ and $V_x \neq V_t$. Note that m is the size of $\mathsf{SKYP}_x(u, v)$. In case (iii), each edge $(u, v)^i (1 \leq i \leq m)$ from u to v represents a skyline path in $\mathsf{SKYP}_x(u, v)$. The optimal query from s to t on $G(V, E)$ is equivalent to the optimal path query on $\bar{G}(\bar{V}, \bar{E})$.

We utilize a *best-first* branch and bound search algorithm as similar as Algorithm 1 to compute the optimal path on $\bar{G}(\bar{V}, \bar{E})$. Note that $\bar{G}$ is not a simple graph, because there are multiple edges between entry u and exit v in a cluster V_x. We define a new search tree as follows such that Algorithm 1 can work on $\bar{G}$.

Given graph $\bar{G}$, starting vertex s and ending vertex t, all possible paths started from s in $\bar{G}$ can be organized in a search tree. Here, the root node represents starting vertex set $\{s\}$. Any non-root node $C = \{v_0, (v_0, v_1), v_1, \cdots, (v_{l-1}, v_l), v_l\}$, represents a path started from s, where $v_0 = s$ and (v_{i-1}, v_i) is an edge from v_{i-1} to v_i. $|C|$ is the number of vertices in C, i.e., $|C| = |\{v | v \in C\}|$. For two different nodes C and C' in the search tree, C is the parent of C' if they satisfy the following two conditions: (i) $C \subset C'$ and $|C'| = |C| + 1$; and (ii) $C' \setminus C$ is a tuple set $\{(u, v), v\}$, where u and v are the ending vertex of path C and C' respectively, and (u, v) is an edge from u to v.

We run Algorithm 1 on this search tree. When a node C pops up from the min-heap H, we expand the C by processing the children of C. Assume that the ending vertex of C is u. Then, for each edge (u, v) in $\bar{G}$, we add tuple set $\{(u, v), v\}$ into C to get a child

	$d = 2$		$d = 3$	
Dataset	Naive index	K-C index	Navie index	K-C index
CAITN	0.0374	0.0078	0.0515	0.0163
CARN	0.0733	0.0217	0.0851	0.0375
EuAll	0.1471	0.0112	0.2019	0.0201
Slashdot	4.8139	0.1321	6.2506	0.1978
HepPh	17.653	0.4372	21.467	0.5402

Table 1: Online Querying Time in Second

	$d = 2$		$d = 3$	
Dataset	Naive index	K-C index	Naive index	K-C index
CAITN	115.99	6.15	203.78	13.31
CARN	2600.68	93.69	4398.95	163.62
EuAll	796.33	20.51	1333.86	38.94
Slashdot	1746.39	46.95	3136.24	81.17
HepPh	4124.96	138.37	6460.35	223.06

Table 2: Index Size in MB

C' of C. There may exist several edges from u to v when $u \in V_x.entry$ and $v \in V_x.exit$. These edges represent skyline paths from u to v in G_x. To check if C' can be pruned or not, we also propose basic pruning rule, skyline path based pruning rule, and LBOP based pruning rule that are as similar as that in Algorithm 1. For LBOP based pruning rule, we compute LBOP as we discussed in Section 4.1. If C' cannot be pruned, it is inserted into min-heap H. Algorithm terminates when $H = \emptyset$ or $f(C) \geq \tau$ for the top element C in H.

6. PERFORMANCE STUDY

All experiments were done on a 3.0GHz Intel Pentium Core i5 CPU PC with 8GB main memory, running on Windows 7.

6.1 Datasets and Experiment Setup

We test the following five real datasets.

CAITN: The **C**hicago **A**nonymized **I**nternet **T**races **N**etwork is a communication network on Chicago. It is an undirected graph with 4,837 vertices and 17,426 edges.

CARN: The **Ca**lifornia **R**oad **N**etwork is an undirect graph with 21,047 vertices and 21,692 edges.

EuAll: EuAll is an email communication network, email users are vertices and the communications between them are edges. It is a directed graph with 11,521 vertices and 32,389 edges.

Slashdot: Slashdot is a technology related news website known for its specific user community. We generate a directed graph with 20,639 vertices and 187,672 edges.

HepPh: HepPh citation graph is a directed graph extracted from the e-print arXiv with 34,546 papers and 421,578 edges.

In each graph, we randomly assigned d costs to each edge ($d \in \{2, 3\}$). we randomly generate 1,000 pairs of vertices and query the shortest paths between each pair of vertices. The query time reported is the average time on each dataset. We set score function as $f(x_1, \cdots, x_d) = \sum_{i=1}^{d} x_i^2$.

6.2 Experimental results

Querying time: As shown in Table 1, we investigate the querying time on five datasets by comparing k-cluster index with naive index for $d = 2$ and $d = 3$. Naive index is the matrix to maintain $\Phi_{u,v}$ for every two vertices u and v in G. In this experiment, we set number of clusters $k = 50$. For all datasets, the querying time of k-cluster index are much less than that of naive index. Specially, in HepPh, the querying time of naive index are 17.653 and 21.467 seconds for $d = 2$ and $d = 3$ respectively but the querying time of k-cluster index are only 0.4372 and 0.5402 seconds. The querying time using k-cluster index are always in order of magnitude faster than naive index. This is because k-cluster index pre-computes the skyline paths for any entry u and any exit v in each cluster.

Index size: The index size is shown in Table 2. We compare the size of k-cluster index with naive index when $d = 2$ and $d = 3$. The number k of clusters is also set as 50. We find the size of k-cluster index are much smaller than that of naive index. In HepPh, for $d = 2$, the size of naive index is 4124.96 MB but the size of k-cluster is only 138.37 MB. These results state k-cluster index is space efficient and thus k-cluster index is suitable for large graphs.

7. RELATED WORK

Most existing works for shortest path problem [3, 6, 7] utilize the property of optimal sub-path in the shortest path: any sub-path of a shortest path is also a shortest path. Therefore, they only need to maintain the shortest paths among vertices in index and compute the shortest path by concatenating the sub shortest paths in index. Unfortunately, in multi-cost graphs, the property of optimal sub-path in a shortest path does not hold. Hence, all these methods cannot solve the optimal path problem proposed in our paper. Mouratidis et al. in [5] studies skyline queries and top-k queries on multi-cost transportation networks. For any vertex v in graph, all distances on different dimensions between v and query point form the cost vector of vertex v. The definition of the cost vector in this work is different with our work and thus its query results are points but not paths. Therefore, the methods in this work cannot be applied to the optimal path problem in our paper.

8. CONCLUSION

In this paper, we defined the optimal path query problem over multi-cost graphs and proposed a *best-first* branch and bound search algorithm with two optimizing strategies. We also proposed a novel index named k-cluster index to make our method more efficient for large graphs. We confirmed the effectiveness and efficiency of our algorithms using real-life datasets.

9. ACKNOWLEDGMENT

This work is supported by the National Grand Fundamental Research 973 Program of China under grant 2012CB316200, the National Natural Science Foundation of China under grant 61173022, 61173023, and grants of the Research Grants Council of the Hong Kong SAR, China No. 419109, 418512.

10. REFERENCES

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006.

[2] S. Chaudhuri and L. Gravano. Evaluating top-*k* selection queries. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB*, pages 397–410. Morgan Kaufmann, 1999.

[3] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD Conference*, pages 457–468, 2012.

[4] N. Ilich and S. P. Simonovic. An evolution program for non-linear transportation problems. *Journal of Heuristics*, 7:145–168, 2001.

[5] K. Mouratidis, Y. Lin, and M. L. Yiu. Preference queries in large multi-cost transportation networks. In *ICDE*, pages 533–544, 2010.

[6] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. In *ICDE*, pages 462–473, 2012.

[7] Y. Xiao, W. Wu, J. Pei, W. Wang, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, pages 493–504, 2009.