Efficient Processing of Which-Edge Questions on Shortest Path Queries*

Petrie Wong[1], Duncan Yung[1], Ming Hay Luk[1], Eric Lo[1], Man Lung Yiu[1], Kenny Q. Zhu[2]

[1] Hong Kong Polytechnic University
{cskfwong, cskwyung, csmhluk, ericlo, csmlyiu}@comp.polyu.edu.hk
[2] Shanghai Jiao Tong University
kzhu@cs.sjtu.edu.cn

Abstract. In this paper, we formulate a novel problem called *Which-Edge* question on shortest path queries. Specifically, this problem aims to find k edges that minimize the total distance for a given set of shortest path queries on a graph. This problem has important applications in logistics, urban planning, and network planning. We show the NP-hardness of the problem, as well as present efficient algorithms that compute highly accurate results in practice. Experimental evaluations are carried out on real datasets and results show that our algorithms are scalable and return high quality solutions.

1 Introduction

Shortest path queries have a wide range of applications in logistics, urban planning, and network planning. This paper introduces a novel problem called *Which-Edge* question on shortest path queries. The objective is to find k edges that minimize the total distance for a given set Q of shortest path queries on a weighted graph $G(V, E, W)$. Let spd denote the total distance of queries in Q on the graph. Specifically, there are two forms of *Which-Edge* questions.

- **Which-k-Edges-Insert question**: Given an edge set P s.t. $P \cap E = \emptyset$, which k edges in P would *minimize* the spd (if those k edges are *inserted* into G)?
- **Which-k-Edges-Delete question**: Given an edge set $\overline{P} \subseteq E$, which k edges in $\overline{P}$ would *minimize* the spd (if they are *removed* from G)?

The potential applications of *Which-Edge* question could be illustrated using a few examples. First, consider an express mail company's delivery network: a graph with nodes representing locations (e.g., cities, warehouses) and edges representing connections (e.g., flights between cities). Subject to rapidly changing business environments, the company may revise its resource allocation policy regularly—with additional resources, the management may want to reduce its overall/average delivery time (increasing its competitive advantage) by establishing a new connection between two indirectly connected locations. In the example above, a *Which-Edge* question natural arises: "*(Q1): which two locations should be connected by the new connection?*". In this situation, answers like "among all the possible choices, the maximum reduction

* This work is supported by the Research Grants Council of Hong Kong (GRF PolyU 520413), and Hong Kong Polytechnic University (ICRG grant A-PL99).

Table 1. Summary of Frequently Used Symbols

Symbol	Meaning
G	the input graph
$z_i(u_i, v_i)$	a bridge to be added to G to connect nodes u_i and v_i
$\|z_i\|$	capacity (resp. length) of a bridge/edge for shortest path
G^{+z_i} / G^{-e_i}	the graph G with bridge z_i added (with edge e_i removed)
P / $\bar{P}$	the set of candidate bridges to add to G (to remove from G)
K	the answer set of a Which-Edge question, containing k bridges/edges
B^{z_i} / D^{e_i}	the benefit (damage) of adding bridge z_i (removing edge e_i)
Z	an arbitrary set of bridges/edges
$sp^G_{(x,y)}$	the *shortest path distance* from x to y on graph G

in overall delivery time (35%) is achieved if a new connection between X and Y is established." will be very helpful in aiding the decision-making process. After said decision is made operational, a new connection (edge) is added to the delivery network (graph) reflecting the impact of the decision. Alternatively, during tough economic times, the management may ask "*(Q2): which k connections, if cut, have the minimal impact on overall delivery time?*" — a decision that is eventually reflected by some existing connections (edges) being removed from the delivery network (graph). Applications of *Which-Edge* questions abound in other domains, too. In urban planning where the road network is modeled as a graph, one may ask "*(Q3): which stretch of roads (edges) should we expand such that average travel times can be significantly reduced?*". In network planning, one may ask "*(Q4): which optic fibre, if broken (e.g., due to a natural disaster), would have the largest negative impact on the average network communication time?*";

Answering *Which-Edge* questions on graphs is an interesting yet challenging research issue. First, we have two types of *Which-Edge* questions, e.g., while Q1 and Q3 are related to *edge insertion*, Q2 and Q4 are related to *edge deletion*. Thus, it is necessary to develop general solutions that are applicable to both questions. Second, the evaluation of *Which-Edge* questions is computationally expensive because the solution space of *Which-Edge* questions can be very large. For example, in Q2, there are many possible k-combinations of connections that could be considered. That huge search space renders straightforward exhaustive algorithms impractical. In this paper, we make the following contributions:

- Introduce the concept of *Which-Edge* questions and present its specification.
- Establish the problem hardness of *Which-Edge* questions.
- Develop evaluation algorithms and efficiency optimizations for answering *Which-Edge* questions.

We will investigate the Which-k-Edges-Insert question and the Which-k-Edges-Delete question in Sections 2 and 3 respectively. In Section 4, we evaluate the solutions' quality and efficiency on real graph datasets. We discuss related work in Section 5 and conclude our paper in Section 6. Table 1 shows a summary of frequently used symbols.

2 Which-k-Edges-Insert Question

In this section, we first formulate the *Which-k-Edges-Insert* question and establish its hardness. Then, we propose our heuristic solutions and efficiency optimizations to compute highly accurate results in practice.

2.1 Problem Formulation

Given a weighted graph $G = (V, E, W)$, there are potentially many non-adjacent node-pairs in G that could be considered to be connected by some new edges. In practice, however, the number of non-adjacent node-pairs to be considered is usually domain-specific. For example in Q1, only flight connections offered by airlines should be considered. Thus, we model the set of non-adjacent node-pairs, P, as an input parameter (generated by other softwares or prepared manually). Specifically, each non-adjacent node-pair (u_i, v_i) in P is associated with an implied *bridge* z_i, which is the edge considered to be added into G for connecting u_i and v_i. The bridge z_i has a length $\|z_i\|$, and a *cost* c^{z_i}, which models the real-world cost of connecting u_i to v_i by z_i in G (e.g., the cost of a bridge can be a function of the bridge's length). In what follows, we use the term "non-adjacent node-pairs" and "bridge" interchangeably.

The *Which-k-Edges-Insert* question aims to find out which k edges in a given bridge-set P, if inserted into G, reduces the sum of shortest path distances of a query workload Q the most (optimization goal). Here we may have a *workload* Q of shortest-path queries with different sources and destinations on G. Each query $q_j(s_j, t_j) \in Q$ is a *distinct* shortest-path query with source s_j and destination t_j. If the user does not specify a workload, we consider Q to contain all-pairs shortest-path queries.

Each query $q_j \in Q$ is associated with an importance factor m_{q_j}—using deliver planning as an example, assuming that only one delivery makes a 100 mile trip from s_1 to t_1, and 50 deliveries make a 5 mile trip from s_2 to t_2, we may model them as two shortest-path queries $q_1(s_1, t_1)$ and $q_2(s_2, t_2)$ and set $m_{q_1} = 1$ and $m_{q_2} = 50$. Thus, query importance can model the number of beneficiaries of a bridge. For instance, a bridge $z_1(u_1, v_1)$, which reduces the shortest-path distance of q_1 from 100 to 40 miles, is not as beneficial as a bridge $z_2(u_2, v_2)$, which reduces the shortest-path distance of q_2 from 5 to 1 mile, because only one delivery makes the trip q_1. More precisely, let $sp^G_{q_j}$ be the shortest-path distance of query q_j on the graph G. Then the benefit of connecting u_i and v_i by z_i on a query $q_j(s_j, t_j)$, or simply the *benefit of bridge z_i on query q_j*, $b^{z_i}_{q_j}$, is the reduction in shortest-path distance of q_j in G^{+z_i} versus G, accounting for the query's importance factor m_{q_j}:

$$b^{z_i}_{q_j} = m_{q_j} \times (sp^G_{q_j} - sp^{G^{+z_i}}_{q_j}) \tag{1}$$

In the example above, the bridge z_1, which shortens q_1 from 100 to 40 miles, has a benefit $b^{z_1}_{q_1} = 1 \times (100 - 40) = 60$; whereas the bridge z_2, which shortens q_2 from 5 miles to 1 mile, has a benefit $b^{z_2}_{q_2} = 50 \times (5 - 1) = 200$.

The above definitions can be extended to a subset K of bridges from P. The *benefit of a set K of bridges on query q_j* is defined as:

$$b^K_{q_j} = m_{q_j} \times (sp^G_{q_j} - sp^{G^{+K}}_{q_j}) \tag{2}$$

The total benefit of K on a query workload Q is:

$$B^K = \sum_{q_j \in Q} b^K_{q_j} \tag{3}$$

PROBLEM 1 (WHICH-k-EDGES-INSERT QUESTION). *Given a graph G, a set P of non-adjacent node pairs (u_i, v_i), their associated bridges z_i and cost c^{z_i}, and a workload of shortest-path queries Q; find a subset $K \subseteq P$ of k bridges so that if they are added to G, they have the maximum benefit to workload Q, accounting of the cost C^K of adding K to G, i.e.,* $\arg\max_{K \subseteq P, |K|=k}(B^K - C^K)$*, where* $C^K = \sum_{z_i \in K} c^{z_i}$.

In this formulation, we assume the cost c^{z_i} of a bridge z_i (e.g., 1000 USD) has been normalized to match the unit of benefit, as in any ranking function in database query processing. In fact, the relationship between the total benefit B^K and the cost C^K is flexible; in some applications we can consider a different formulation, for example, using another function $\arg\max_{K \subseteq P, |K|=k}(B^K / C^K)$.

2.2 Problem Hardness

We prove that this problem is $\mathcal{NP}$-hard.

Theorem 1. *The Which-k-Edges-Insert problem is $\mathcal{NP}$-hard.*

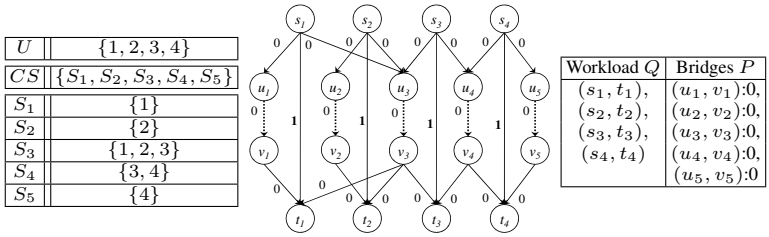
Proof. We present a *reduction scheme* that converts any given instance of the *Set-Cover problem* [1] into an instance $\langle k, G(V, E), P, Q\rangle$ of our Which-k-Edges-Insert problem. Let $\langle k, CS = \{S_i\}, U\rangle$ be an instance of Set-Cover, where k is an integer, U is a domain set of items, CS is a collection of subsets $S_i \subseteq U$. This problem asks whether there exists a sized-k collection $CS' \subseteq CS$ such that the size of its subset union $|\cup_{S_i \in CS'} S_i|$ equals to $|U|$. The reduction scheme is as follows:

- for each item $j \in U$, we insert a query $q_j(s_j, t_j)$ into Q, and insert the vertices s_j, t_j into V;
- for each subset $S_i \in CS$, we insert a directed bridge $z_i(u_i, v_i)$ with length 0 into P and insert the vertices u_i, v_i into V;
- for each query $q_j(s_j, t_j)$ of Q, we insert a directed edge (s_j, t_j) with length 1 into E;
- for each item $j \in U$ in a subset $S_i \in CS$, we insert directed edges (s_j, u_i) and (v_i, t_j) with length 0 into E.

An example reduction is illustrated in Figure 1. The bridges in P are shown as dashed lines. Observe that the size of the constructed instance $\langle k, G(V, E), P, Q\rangle$ is polynomial to the size of the given instance $\langle k, CS = \{S_i\}, U\rangle$. Also, the construction process takes polynomial time. The intuition behind this reduction scheme is that, if a bridge z_i is selected in a solution of Which-k-Edges-Insert, then the queries q_j benefit from it correspond to the items j covered by a chosen set S_i in a solution of Set-Cover.

Now, we only consider the subclass $\mathcal{C}$ of problem instances of Which-k-Edges-Insert that conform with the conditions in the above reduction scheme, specifically:

- all vertices in P and Q are unique; they are the only vertices in the graph G;
- each bridge $z_i(u_i, v_i) \in P$ has length 0;
- for each query $q_j(s_j, t_j) \in Q$, there must be an edge (s_j, t_j) with length 1 in the graph;
- in addition, the graph contains only the following edges: an edge (s_j, u_i) exists if and only if an edge (v_i, t_j) exists (for some z_i, q_j); such edges (if exist) must have length 0;



(a) Set-Cover (b) Which-k-Edges-Insert: graph G (c) content of Q and P

Fig. 1. Reduction: Set Cover to Which-k-Edges-Insert

Now, we show that a solution of Set-Cover $\langle k, CS = \{S_i\}, U\rangle$ corresponds to a solution of the $\mathcal{C}$ subclass of Which-k-Edges-Insert problem $\langle k, G(V, E), P, Q\rangle$ with benefit equal to $|Q|$, and vice versa.

Let CS' be a solution of the Set-Cover. Let K' be a solution of the Which-k-Edges-Insert problem. Let $M = |U| = |Q|$.

We first convert a given solution CS' to a corresponding solution K' and then derive the benefit value of K'. Let $U' = \cup_{S_i \in CS'} S_i$ be the union set of items covered by CS'. Recall that the size of CS' is k, and the size of U' is M. WLOG, we rename CS' as $\{S_{x_1}, S_{x_2}, \cdots, S_{x_k}\}$ and rename U' as $\{y_1, y_2, \cdots, y_M\}$. We claim that the corresponding solution of the Which-k-Edges-Insert problem is $K' = \{z_{x_1}, z_{x_2}, \cdots, z_{x_k}\}$, and the set of benefited queries is $Q' = \{q_{y_1}, q_{y_2}, \cdots, q_{y_M}\}$. Since CS' is a solution of the Set-Cover, each item in $y_j \in U'$ must be contained in S_{x_i} for some i. According to our reduction scheme, the corresponding edges (s_j, u_i) and (v_i, t_j) belong to the constructed graph. Thus, for each query $(s_j, t_j) \in Q'$, there exists a path s_j, u_i, v_i, t_j with distance 0, causing its benefit to be 1. Summing the benefit over all queries of Q', we obtain the benefit M.

We then convert a given solution K' to a corresponding solution CS' and then derive the union size of CS'. Since the size of CS' is k, we rename K' as $K' = \{z_{x_1}, z_{x_2}, \cdots, z_{x_k}\}$. The benefit of each query is either 0 or 1. Since the total benefit of K' on all queries is M, there must be a set of M benefited queries, say, $Q' = \{q_{y_1}, q_{y_2}, \cdots, q_{y_M}\}$. We claim that the corresponding solution of the Set-Cover problem is $CS' = \{S_{x_1}, S_{x_2}, \cdots, S_{x_k}\}$, with the union set of items covered as $U' = \{y_1, y_2, \cdots, y_M\}$. Since K' is a solution of the Which-k-Edges-Insert problem, each query in $q_{y_j} \in Q'$ must be benefited by the bridge z_{x_i} for some i. According to our reduction scheme, the corresponding item in $y_j \in U'$ must be contained in S_{x_i} for some i. Thus, all items in U' are covered by CS' and the union size is M.

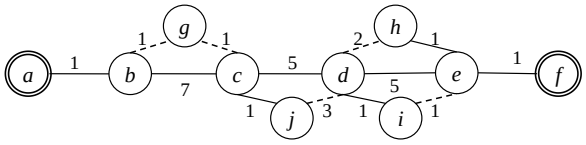
Since the Set-Cover problem is $\mathcal{NP}$-hard [1], the above implies that Which-k-Edges-Insert problem is also $\mathcal{NP}$-hard. □

Observe that Set-Cover instances correspond to only a subclass of problem instances of Which-k-Edges-Insert. Thus, the approximation algorithms (and their approximation ratio) for Set-Cover cannot be applied to all problem instances of Which-k-Edges-Insert.

Brute-Force Solution We proceed to describe a brute-force algorithm (BF) for computing the exact result for the Which-k-Edges-Insert problem. It enumerates every possible

subset K with k bridges from P. For each subset K, it temporarily inserts bridges in K to the graph G (denote that as G^{+K}) and uses an incremental shortest path algorithm (e.g., [2]) to compute the new shortest path. Finally, the subset K with the highest benefit B^K is reported as the result. Figure 2a shows an example[3] with k=2, a source s, a sink t, and with shortest path distance originally as 1. The running steps of BF with $P = \{(b, g), (g, c), (d, h)\ , (j, d), (i, e)\}$ are illustrated in Figure 2b. There are $\binom{|P|}{k} = \binom{5}{2}$ subsets to be considered. The subset $K = \{(b, g), (g, c)\}$ is reported as the result because the new shortest path distance is 14 after adding them to G, which yields the highest benefit $B^K = 19 - 14 = 5$ among other subsets.

The time complexity of BF is $\mathcal{O}(\binom{|P|}{k} ISP(G))$, where $ISP(G)$ denotes the time complexity of an incremental shortest path algorithm. Note that an incremental shortest path algorithm [2] takes $\mathcal{O}(\log |V|)$ time. Furthermore, observe that the time complexity $\mathcal{O}(\binom{|P|}{k} ISP(G))$ is exponential to k. Therefore, the term $\binom{|P|}{k}$ renders BF only feasible for a tiny $|P|$.



(a) graph

K	sp^{G+K}	B^K
$(\mathbf{b}, \mathbf{g}), (\mathbf{g}, \mathbf{c})$	**14**	**5**
$(b, g), (d, h)$	17	2
$(b, g), (i, e)$	16	3
$(b, g), (j, d)$	18	1
$(d, h), (g, c)$	17	2
$(d, h), (i, e)$	16	3
$(d, h), (j, d)$	16	3
$(g, c), (i, e)$	16	3
$(g, c), (j, d)$	18	2
$(i, e), (j, d)$	15	4

(b) Brute-Force

↙ Optimal answer: K={$(b, g), (g, c)$}, B^K=5

z_i	Iteration 1		Iteration 2	
	sp^{z_i}	B^{z_i}	sp^{z_i}	B^{z_i}
(b, g)	19	19-19=0	16	16-16=0
(d, h)	17	19-17=2	16	16-16=0
(g, c)	19	19-19=0	16	16-16=0
(i, e)	**16**	**19-16=3**	-	-
(j, d)	18	19-18=1	15	**16-15=1**

Greedy answer: K={$(i, e), (j, d)$}, B^K=4

TopK answer: K={$(i, e), (d, h)$}, B^K=3

(c) Greedy and TopK

Fig. 2. Which-k-Edges-Insert Question: Running Steps of Algorithms (k=2)

2.3 Heuristics Solutions

We now present two polynomial-time heuristics algorithms that return a subset $K \subseteq P$ of k bridges whose benefit is an heuristics of the optimal benefit. Empirical results show that the algorithms can return high quality solutions using a very reasonable amount of time.

Both algorithms require a function called `CalSPBenefit`. Its goal is to compute the benefit B^{z_i} of every *single* (note: not combination of) bridge $z_i \in P$. A naive implementation of the function `CalSPBenefit` can be based on a nested-loop:

[3] For ease of illustration, we assume bridge costs being 0 in the example. In fact, our solutions can deal with arbitrary values of c^{z_i}.

Algorithm 1 Algorithm SP-NL-Benefit

```
1: function SP-NL-BENEFIT(G, Q, P) implements CalSPBenefit
2:     for each bridge z_i of P do                                  ▷ outer loop
3:         for each q_j ∈ Q do                                      ▷ inner loop
4:             IncSP(q_j, G^{+z_i})
5:             Calculate the benefit b_{q_j}^{z_i} of z_i
6:         B^{z_i} = Σ_{q_j ∈ Q} b_{q_j}^{z_i}                      ▷ Equation 3
```

SP-NL-Benefit has a complexity of $\mathcal{O}(|Q||P|ISP(G))$, where $ISP(G)$ is the time complexity of an incremental shortest path algorithm `IncSP` (e.g,. $\mathcal{O}(\log|V|)$ [2]). So, SP-NL-Benefit is not efficient enough because it invokes `IncSP` $|Q| \times |P|$ times. In Section 2.4, we will present more efficient implementations for `CalSPBenefit`. Now, we present the two heuristics algorithms first.

a) Greedy Algorithm Our first heuristics algorithm is based on greedy heuristics. Algorithm 2 shows the pseudo-code of our Greedy algorithm. It invokes the function `CalSPBenefit` in k iterations. In each iteration, it greedily selects the bridge $z^* \in P$ with the highest benefit B^{z^*}, removes it from P, and then inserts it into G. The time complexity of Greedy is $\mathcal{O}(k \cdot CB(G,P))$, where $CB(G,P))$ denotes the time complexity of an implementation of function `CalSPBenefit`. Even using the (slow) naive implementation of `CalSPBenefit` (see Algorithm 1), Greedy is still a polynomial time algorithm.

Algorithm 2 Heuristics Algorithm (Greedy)

```
1: function GREEDY(G, s, t, P, k)
2:     K = ∅                                                        ▷ result set
3:     while |K| < k do
4:         CalSPBenefit(G, s, t, P)
5:         let z* ∈ P be the bridge with the highest benefit B^{z*}
6:         remove z* from P
7:         insert z* into K                                         ▷ z* is one of the selected bridges
8:         G = G ∪ z*                                               ▷ update G to include z* as well
9:     return the result set K
```

The running steps of Greedy on the example in Figure 2a are illustrated in Figure 2c. For $k = 2$, Greedy has two iterations. In the first iteration, Greedy first invokes the function `CalSPBenefit` to find the best bridge (i,e), which has a benefit of 3. So it is removed from P and gets inserted into the final result set K and into the graph G. In the second iteration, function `CalSPBenefit` is invoked the second time. Note that the benefit of some bridges (e.g., (d,h)) may change after the graph has become $G^{+(i,e)}$. Again, the bridge with the highest benefit in the second iteration (i.e., (j,d)) is inserted into the final result set K. Since $k = 2$, Greedy stops after this iteration. Greedy finds the optimal result $\{(i,e),(j,d)\}$ in this example.

b) TopK Algorithm Our second heuristics algorithm, called TopK (Algorithm 3), attempts to further trade the result quality for better efficiency. It simply executes the function `CalSPBenefit` once and then returns the top-k most beneficial bridges. Its time complexity is $\mathcal{O}(CB(G, P))$, i.e., the complexity of function `CalSPBenefit`.

Algorithm 3 Heuristics Algorithm (TopK)

1: **function** TOPK(G, s, t, P, k)
2: `CalSPBenefit`(G, s, t, P)
3: $K =$ the k most beneficial bridges ▷ result set
4: **return** the result set K

The running steps of TopK on the example in Figure 2a is the same as the first iteration of Greedy, as illustrated in Figure 2c. However, TopK has only one iteration. It triggers function `CalBenefit` once, and returns the top-2 bridges (i, e) and (d, h) as the result set. The total benefit of (i, e) and (d, h) is $19 - 16 = 3$.

2.4 Efficiency Optimization

(a) Pruning Candidate Bridges The trick is very simple: if the length $\|z_i\|$ of a bridge z_i is longer than all queries' shortest-path distances, such bridge can be pruned right away because it yields no benefit. Furthermore, we skip a query q_j's `IncSP` call if its original shortest-path distance is smaller than the lengths of all the given candidate bridges.

(b) Input-Adaptive Benefit Calculation The following lemmas enable us to determine the benefits of all bridges simply by (i) invoking `IncSP` (using each u_i or v_i as the source) $|P|$ times or (ii) invoking `IncSP` (using each s_j *and* each t_j as the sources) $2|Q|$ times. Therefore, our proposed algorithm, namely, SP-Fast-Benefit, implements `CalSPBenefit`, by doing (i) when $|P| < 2|Q|$ or doing (ii) otherwise. Briefly, in Lemma 1, by executing `IncSP` from a node u_i, we can obtain the shortest-path tree of u_i, which embeds the shortest-path distance of all nodes from u_i. For the path s_j to t_j via u_i to be shorter than $sp^G_{q_j}$, either the path s_j to u_i, or the path u_i to t_j must contain the node v_i and hence the bridge (u_i, v_i). So, we will have enough distance information to calculate the updated shortest path distances of all queries in Q.

Lemma 1. *Given a bridge $z_i(u_i, v_i)$, a graph G, and a query workload Q, the benefit B^{z_i} of bridge z_i can be obtained by executing `IncSP` only once, using either u_i or v_i as the source, and stops when all nodes in Q are seen.*

Proof. Assume z_i is inserted to G, so the graph becomes G^{+z_i}. If the shortest path of a query $q_j(s_j, t_j) \in Q$ on G^{+z_i} is shorter than its original shortest path on G, the new shortest path of q_j on G^{+z_i} must pass through z_i and its corresponding (new and shorter) shortest path distance $sp^{G^{+z_i}}_{q_j}$ would be:

$$sp^{G^{+z_i}}_{q_j(s_j,t_j)} = sp^{G^{+z_i}}_{(s_j,u_i)} + sp^{G^{+z_i}}_{(u_i,t_j)} \tag{4}$$

where $sp^{G^{+z_i}}_{(s_j,u_i)}$ and $sp^{G^{+z_i}}_{(u_i,t_j)}$ are the shortest path distances from u_i to s_j and t_j, respectively.

Given $sp^{G^{+z_i}}_{q_j(s_j,t_j)}$, we derive the benefit $b^{z_i}_{q_j}$ of z_i on query q_j as:

$$b^{z_i}_{q_j} = m_{q_j} \times (sp^{G}_{q_j} - \min(sp^{G}_{q_j}, sp^{G^{+z_i}}_{q_j(s_j,t_j)})) \tag{5}$$

As m_{q_j} and $sp^{G}_{q_j}$ are given, from Equations 4 and 5, we can see that as long as we have the values of $sp^{G^{+z_i}}_{(s_j,u_i)}$ and $sp^{G^{+z_i}}_{(u_i,t_j)}$, we are able to compute $b^{z_i}_{q_j}$. Obviously, given a graph G^{+z_i} with bridge $z_i(u_i, v_i)$ inserted, we can obtain the shortest path distances from u_i to all nodes using one Dijkstra's execution. After that, shortest path distances like $sp^{G^{+z_i}}_{(s_j,u_i)}$ and $sp^{G^{+z_i}}_{(u_i,t_j)}$ are readily available if we make sure the shortest-path execution stops only when all source nodes s_j and destination nodes t_j in the workload Q are seen. By repeatedly applying Equation 5 using the information above, all benefits $b^{z_i}_{q_j}$ can be obtained and the overall benefit B^{z_i} of bridge z_i on workload Q can be derived using Equation 3. The proof is the same if we use v_i as the source instead of u_i. □

Lemma 2. *Given a query $q_j(s_j, t_j)$, a graph G, a set of bridges P, the benefit $b^{z_i}_{q_j}$ for all bridges $z_i(u_i, v_i)$ in P with respect to a query q_j in Q can be obtained by executing* `IncSP` *twice, once using s_j as the source, once using t_j as the source, and stop when all nodes in P are seen.*

Proof. If the shortest path of a query $q_j(s_j, t_j) \in Q$ on G^{+z_i} is shorter than its original shortest path on G, the new shortest path of q_j on G^{+z_i} must pass through $z_i(u_i, v_i)$ and its corresponding (new and shorter) shortest path distance $sp^{G^{+z_i}}_{q_j}$ would be:

$$\begin{aligned} sp^{G^{+z_i}}_{q_j(s_j,t_j)} = \min(\ & sp^{G^{+z_i}}_{(s_j,u_i)} + ||(u_i, v_i)|| + sp^{G^{+z_i}}_{(v_i,t_j)}, \\ & sp^{G^{+z_i}}_{(s_j,v_i)} + ||(u_i, v_i)|| + sp^{G^{+z_i}}_{(u_i,t_j)}) \end{aligned} \tag{6}$$

where $||(u_i, v_i)||$ denote the length of edge (u_i, v_i).

Given $sp^{G^{+z_i}}_{q_j(s_j,t_j)}$, we can derive the benefit $b^{z_i}_{q_j}$ of z_i on query q_j using Equation 5. Since $||(u_i, v_i)||$, m_{q_j}, and $sp^{G}_{q_j}$ are given, from Equations 5 and 6, by using the values of $sp^{G^{+z_i}}_{(s_j,u_i)}$, $sp^{G^{+z_i}}_{(s_j,v_i)}$, $sp^{G^{+z_i}}_{(v_i,t_j)}$, and $sp^{G^{+z_i}}_{(u_i,t_j)}$, we are able to compute $b^{z_i}_{q_j}$. Obviously, given a graph G, we can obtain the shortest path distances from s_j and t_j to all nodes using two shortest-path executions. After that, shortest path distances like $sp^{G^{+z_i}}_{(s_j,u_i)}$, $sp^{G^{+z_i}}_{(s_j,v_i)}$, $sp^{G^{+z_i}}_{(v_i,t_j)}$, and $sp^{G^{+z_i}}_{(u_i,t_j)}$ are readily available if we make sure the shortest-path executions stop only when all nodes u_i and v_i in the P are seen. By repeatedly applying Equation 6 using the information above, the benefits $b^{z_i}_{q_j}$ of all bridges on a query $q_j(s_j, t_j)$ can be obtained. □

3 Which-k-Edges-Delete Question

Next, we formulate the *Which-k-Edges-Delete* question and establish its hardness. Then, we adapt our heuristic solutions from the previous section to answer the *Which-k-Edges-Delete* question.

3.1 Problem Formulation

This problem aims to find out which k edges in a given set $\bar{P} \subseteq E$, if deleted, have the least impact on the overall shortest-path distances (e.g., Q2). In this case, the "damage" $D^{e_i}_{q_j}$ of deleting an edge e_i with respect to a query workload Q can be defined as:

$$D^{e_i} = \sum_{q_j \in Q} m_{q_j}(sp^{G^{-e_i}}_{q_j} - sp^{G}_{q_j}) \tag{7}$$

where $sp^{G^{-e_i}}_{q_j}$ denotes the shortest path of q_j on a graph G without edge e_i.

3.2 Problem Hardness and Solutions

Observe that an instance $\langle k, G, P, Q\rangle$ of the Which-k-Edges-Insert problem is equivalent to an instance $\langle |\bar{P}| - k, G^{+\bar{P}}, \bar{P}, Q\rangle$ of the Which-k-Edges-Delete problem, where $\bar{P} = P$. Since the Which-k-Edges-Insert problem is $\mathcal{NP}$-hard, the Which-k-Edges-Delete problem is also $\mathcal{NP}$-hard.

All our solutions (BF, Greedy, and TopK) are applicable here. First, we can also have a naive nested-loop implementation method for a function, `CalSPDamage`, that calculates the damage of deleting each edge in $\bar{P}$ (the set of candidate edges to be removed from G) on the query workload Q by calling a shortest incremental algorithm. Then for Greedy, we greedily choose the edge e with the least damage, delete e from G, and repeat k iterations. For TopK, we simply choose the k-lowest damage edges in the first iteration.

3.3 Efficiency Optimization

The problem of `CalSPDamage` is: Given a subset $\bar{P}$ of edges in E, calculate the damage D^{e_i} of each edge $e_i \in \bar{P}$ with respect to a query workload Q. A nested-loop algorithm, SP-NL-Damage, that invokes `IncSP` $|Q| \times |P|$ times is also applicable here.

To develop a more efficient algorithm. The question is how to obtain the item $sp^{G^{-e_i}}_{q_j}$ for all queries $q_j \in Q$ in Equation 7 efficiently. Observe that the deletion of an edge e_i from G is equivalent to the insertion of the edges $\bar{P} - \{e_i\}$ into the graph $G^{-\bar{P}}$, i.e., $G^{-e_i} = G^{-\bar{P}+(\bar{P}-e_i)}$. Note that $sp^{G^{-e_i}}_{q_j}$ means the shortest path distance of query q_j on the graph G without e_i. In other words, it is the shortest path distance of query q_j on the graph $G^{-\bar{P}+(\bar{P}-e_i)}$.

Suppose that $\bar{P} = \{e_1(u_1, v_1), e_2(u_2, v_2), e_3(u_3, v_3)\}$. In this example, the graph without e_1, i.e., G^{-e_1}, is the same as the graph without $\bar{P}$, but with edges e_2 and e_3 inserted back, i.e., $G^{-e_1} = G^{-\bar{P}+\{e_2,e_3\}}$. Therefore, if we want to compute the changes of shortest path values (indirectly, the damage) of deleting e_1 from G, we can consider the changes of shortest path values (indirectly, the benefit) of *inserting* e_2 and e_3 to $G^{-\bar{P}}$.

From the discussion above, our idea is to solve the deletion problem as an insertion problem on the graph $G^{-\bar{P}}$. Instead of using the entire $G^{-\bar{P}}$, we observe that the following shortest paths on $G^{-\bar{P}}$ are essential for solving the insertion problem on the graph $G^{-\bar{P}}$. Specifically, the following paths are necessary for computing shortest paths $sp^{G_{e_i}}_{q_j}$ for any edge e_i and query q_j.

– for each query $q_j(s_j, t_j)$, the shortest path $s_j \rightsquigarrow t_j$;
– for each combination of edges $e_i(u_i, v_i)$ and $e_h(u_h, v_h)$, the shortest paths $u_i \rightsquigarrow u_h$, $u_i \rightsquigarrow v_h$, $v_i \rightsquigarrow u_h$, $v_i \rightsquigarrow v_h$;
– for each combination of edge $e_i(u_i, v_i)$ and query $q_j(s_j, t_j)$, the shortest paths $u_i \rightsquigarrow s_j$, $u_i \rightsquigarrow t_j$, $v_i \rightsquigarrow s_j$, $v_i \rightsquigarrow t_j$.

Based on the above observation, we develop an efficient algorithm, SP-Fast-Damage, to create a *reduced graph* $\mathcal{G}$ that contains only nodes and edges involved in the shortest paths above. The computation is efficient because it works on the concise $\mathcal{G}$.

4 Experiments

In this section, we present experiment results based on real graph data. All experiments were run on a 2.5 GHz Intel PC running Ubuntu with 8 GB of RAM. We evaluated the algorithms by running experiments on four real undirected graphs of different sizes and types (Figure 3). The queries in Q are selected randomly. The bridge set P (for insertion) and the edge set $\bar{P}$ (for deletion) are selected randomly. Here, we also present results of both edge insertion case and edge deletion case only. The default bridge cost and query importance are 0 and 1, respectively. Experimental results using other values are largely similar, so we do not present them here.

Undirected Graphs (for shortest-path)	node Count	Edge Count	Avg Degree
Argentina Road Network (ARG) http://www.maproom.psu.edu/dcw	85,287	88,357	2.07
San Francisco Road Network (SF) http://www.maproom.psu.edu/dcw	174,956	223,001	2.54
CAIDA Internet Router Topology (LINKS) http://www.caida.org/tools/measurement/skitter	190,914	607,609	6.36

Fig. 3. Real Graph Data Used in Experiments

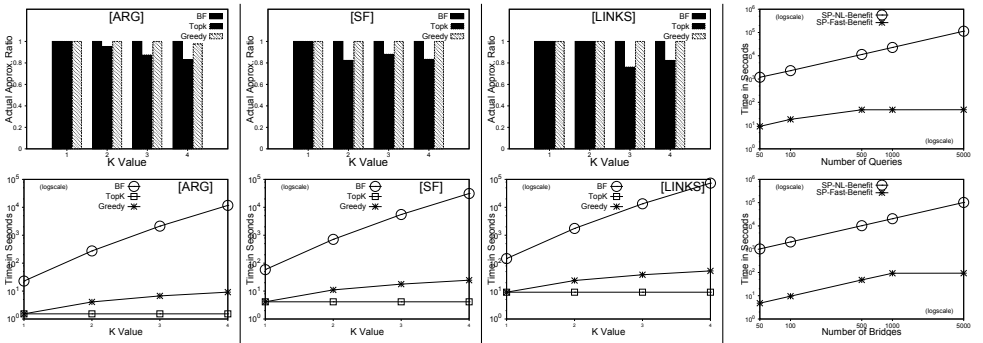
4.1 Which-k-Edges-Insert Question

The first row of Figures 4(a)–(c) shows the actual approximation ratio (A.A.R.) of BF, Greedy, and TopK on four real graphs. In this experiment, we limit the experiment setting to $|P| = |Q| = 25$ to let BF compute the optimal solution in a reasonable time. BF is an exact but exponential algorithm, it always has a ratio of 1. On datasets SF and LINKS, Greedy returns the optimal solution. On dataset ARG, the A.A.R. of Greedy degrades a little bit when $k = 4$, but it still returns 95% of the optimal benefit. The solutions of TopK are quite data dependent. On ARG, its A.A.R decreases when k increases. On SF, its A.A.R. is from 0.8 to 1. On LINKS, its A.A.R. drops initially when k increases but it rises again later.

The second row of Figures 4(a)-(c) shows the running times of BF, Greedy, and TopK. As a baseline, BF takes up to about a day when k is four on large graphs like SF and LINKS, even $|P|$ and $|Q|$ are so small ($= 25$) in this experiment. TopK and Greedy are orders of magnitude faster than BF and Greedy is about k times slower than TopK.

As the performances of Greedy and TopK mainly depend on the implementations of the `CalSPBenefit`, we evaluate the performance of the two implementations of

`CalSPBenefit` function: SP-NL-Benefit and SP-Fast-Benefit, by varying $|P|$ and $|Q|$. Figure 4(d) shows their running times when we vary the size of workload ($|Q| = 50$ to 5000; default $|P| = 500$) and the size of bridge sets ($|P| = 50$ to 5000; default $|Q| = 500$) on the ARG dataset. The results on the other three datasets are largely similar to the results on ARG, so we do not present them here. From the results we can see that SP-NL-Benefit is about two orders of magnitude slower than SP-Fast-Benefit. When fixing $|P|$ at 500 and varying $|Q|$ (Figure 4(d); top), the running time of SP-Fast-Benefit goes flat at $|Q| = 500$ because it decides to invoke `IncSP` $|P|$ times at that point. Similarly, when fixing $|Q|$ at 500 and varying $|P|$ (Figure 4(d); bottom), the running time of SP-Fast-Benefit goes flat at $|P| = 1000$ because it decides to invoke `IncSP` $2|Q|$ times at that point.



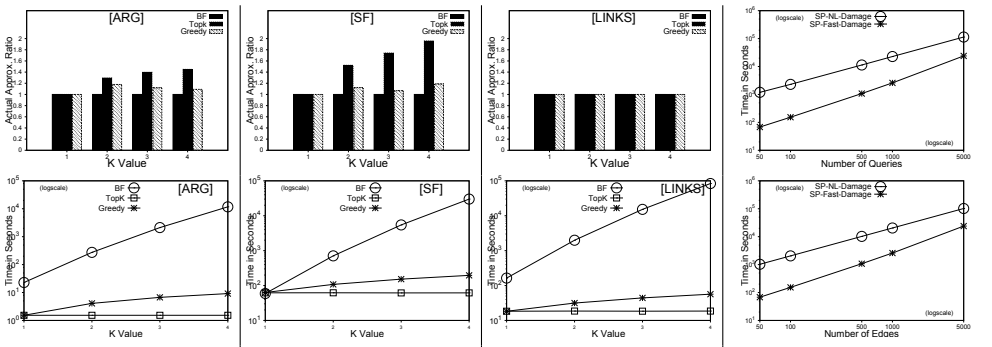
(a) [ARG] vary k (b) [SF] vary k (c) [LINKS] vary k (d) [ARG] vary $|Q|$ and $|P|$

Fig. 4. Which-k-Edges-Insert Question

4.2 Which-k-Edges-Delete Question

Figures 5(a)–(c) show the A.A.R. and the performance of BF, Greedy, and TopK, for the case of edge deletion ($|P| = |Q| = 25$). The 1.0 A.A.R of BF is put there as reference. Both Greedy and TopK found the same optimal solutions as BF and LINKS. On ARG and SF, the worst A.A.R of Greedy is 1.2 (meaning the damage of Greedy is 20% more than the optimal), which is a good result. The worst A.A.R 1.9 of TopK (at SF, k=4) is also low. TopK runs almost k times faster than Greedy. Both of them are orders of magnitude faster than BF.

Figure 5(d) shows the running times of SP-NL-Damage and SP-Fast-Damage, two implementations of `CalSPDamage`, under different workload size ($|Q| = 50$ to 5000) and edge set size ($|P| = 50$ to 5000), on ARG. The results on the other three datasets are largely similar to the results here, so we do not present them here. From the results we can see that both SP-NL-Damage and SP-Fast-Damage scale well but SP-NL-Damage is an order of magnitude slower than SP-Fast-Damage. We have also plotted a scalability graph showing the running times of SP-NL-Damage and SP-Fast-Damage on the four real graphs ($|Q| = |P| = 500$). Results show that they are both scalable to graphs of different sizes.



(a) [ARG] vary k (b) [SF] vary k (c) [LINKS] vary k (d) [ARG] vary $|Q|$ and $|P|$

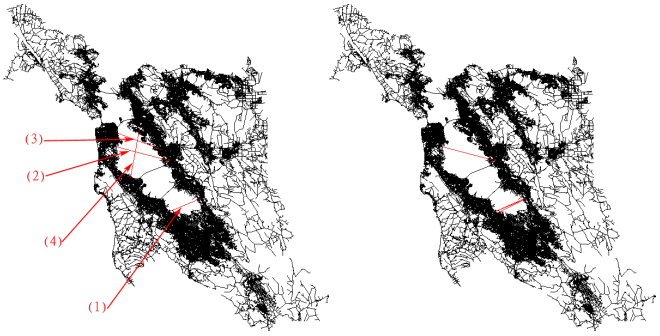
Fig. 5. Which-k-Edges-Delete Question

4.3 Case Study

Here we present the findings of a case study of applying a Which-k-Edges-Insert question on the San Francisco Bay area road network. The goal is to determine the best locations for constructing a new bridge spanning the San Francisco Bay such that travel distances can be reduced the most (i.e., Q3 in Section 1). In the study, we concerned only with the construction of new bridges spanning the San Francisco Bay. We identified a number of possible locations for constructing new bridges, which resulted in 348 possible bridges to consider constructing. For all 348 candidate bridges, we assumed that their construction costs are directly related to the bridge's span length; thus, longer bridges are more costly to construct, but may shorten travel distances more, when compared with shorter bridges. To determine important travel destinations, we used San Francisco Bay Area commuter statistics supplied by the Metropolitan Transportation Commission. From the Metropolitan Transportation Commission, we mapped county to county commuter statistics back onto San Francisco Bay Area road network graph. After remapping the county to county commuter statistics, we identified 377 queries to characterize the commuter data supplied from the Metropolitan Transportation Commission. For each of these 377 commuter queries, their relative importance to San Francisco Bay Area residents were computed from census data[4], which were collected by the Metropolitan Transportation Commission.

Figure 6 shows the original San Francisco Bay Area road network (in black color). Using Greedy, the four most beneficial bridges to add into the San Francisco Bay Area road network are shown in Figure 6a (red colors): the single most beneficial bridge is shown just south of the Dumbarton Bridge (annotated with (1) in the figure). This bridge would help commuters traveling between the extreme ends of the Silicon Valley at the southern end of Alameda County and the southern end of San Mateo County. The second most beneficial bridge to construct is annotated with (2) in the figure. This bridge connects south San Francisco to the mid southern end of Alameda County. The third most beneficial bridge (annotated with (3)), connects central San Francisco to central Alameda County. Finally, the fourth most beneficial bridge (annotated with (4)) to construct connects Alameda County to San Mateo County. Figure 6b shows the four bridges suggested by TopK. We can see that TopK selects to construct 3 bridges all

[4] http://www.mtc.ca.gov/maps_and_data/datamart/census/county2county/table1coco.html



(a) Greedy ($k = 4$) (b) TopK ($k = 4$)

Fig. 6. Case Study [***please view this figure from color PDF file or color print copy***]

south of the Dumbarton Edge in close proximity to each other. The running time of TopK algorithm is about 10 seconds, where Greedy is about 40 seconds, 4 times slower than TopK. Both running times are highly reasonable. However, the bridges suggested by Greedy are more insightful than bridges suggested TopK in this case study.

5 Related Work

Decision support and data mining on graphs (e.g., [3–7]) are related to this work. However, they have different focuses (e.g., summarizing, OLAP, or mining graphs) with us.

The broad applications of *Which-Edge* query makes it relevant to a number of other areas: spatial database, operations research, dynamic graph maintenance, and detection of high risk network links.

Optimal-location queries [8–10] are a class of spatial decision-support queries where users look for the best location, l, for a new facility such that the greatest *benefit* is obtained. For example, [9] considers the benefit of a location as the total weight of its reverse nearest neighbors (i.e., the total weight of objects that are closer to l than to any other data point in the dataset). Optimal-location queries are helpful in finding ideal locations for a new shop to attract the largest number of customers. However, *Which-Edge* queries (e.g., Q1–Q4) that work on graphs are more diverse than optimal-location queries, which only ask "*where to add a new point?*"

The reverse optimization problems [11] in operations research are relevant to us. An inverse optimization problem takes a *feasible solution* x as input and then tunes the problem's parameters, with as low of a cost as possible, such that x becomes the optimal solution. In reverse optimization problems, a *target value* v is specified, and then the problem's parameters are tuned, with as low cost as possible, such that v becomes either the optimal value or an upper bound of the optimal value for the problem. For example, in *reverse shortest-path problems* [11], an input of the desired shortest-path distance d to a shortest-path query q yields an output of a set of edge weight adjustments that make the shortest-path distance of q shorter than d. These existing operations research problems require the user to *explicitly state* the target to be tuned (e.g., the desired shortest-path distance d). In contrast, *Which-Edge* queries *tell* the user the target's identity and its optimized value.

If the set of graph elements that will be changed is explicitly known, then it is related to the dynamic graph maintenance (e.g., [12, 13, 2]) problems, whose goals are to efficiently update the graph measures (a.k.a. incremental update). However, those works do not tell users which set of graph elements is worthwhile to get updated, which is the objective of this work. Most network maintenance works (e.g., [14]) aim to localize faulty links *after* some links break. *Which-Edge* queries, like Q4, can be used to determine critical links in a network such that effective preventive maintenance measures can be implemented *before* any link breaks.

6 Conclusion

In this paper, we formulate a novel *Which-Edge* question on shortest path queries. It has important applications in logistics, urban planning, and network planning. We show the NP-hardness of the problem, as well as present efficient algorithms with optimizations for computing highly accurate results in practice. In future, we will investigate extensions of *Which-Edge* question for other graph queries (e.g., reachability queries).

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2001)
2. Buriol, L.S., Resende, M.G.C., Thorup, M.: Speeding up dynamic shortest-path algorithms. INFORMS Journal on Computing **20**(2) (2008) 191–204
3. Chen, C., Lin, C.X., Fredrikson, M., Christodorescu, M., Yan, X., Han, J.: Mining graph patterns efficiently via randomized summaries. PVLDB **2**(1) (2009) 742–753
4. Chen, C., Yan, X., Zhu, F., Han, J., Yu, P.S.: Graph OLAP: Towards Online Analytical Processing on Graphs. In: ICDM. (2008) 103–112
5. Khan, A., Yan, X., Wu, K.L.: Towards proximity pattern mining in large graphs. In: SIGMOD. (2010) 867–878
6. Tian, Y., Hankins, R.A., Patel, J.M.: Efficient aggregation for graph summarization. In: SIGMOD. (2008) 567–580
7. Zhang, N., Tian, Y., Patel, J.M.: Discovery-driven graph summarization. In: ICDE. (2010) 880–891
8. Du, Y., Zhang, D., Xia, T.: The optimal-location query. In: SSTD. (2005) 163–180
9. Gao, Y., Zheng, B., Chen, G., Li, Q.: Optimal-location-selection query processing in spatial databases. TKDE **21** (2009) 1162–1177
10. Xiao, X., Yao, B., Li, F.: Optimal location queries in road network databases. In: ICDE. (2011) 804–815
11. Zhang, J., Lin, Y.: Computation of reverse shortest-path problem. Journal of Global Optimization **25** (2003) 243–261
12. Demetrescu, C., Italiano, G.F.: Algorithmic techniques for maintaining shortest routes in dynamic networks. Electron. Notes Theor. Comput. Sci. **171** (April 2007) 3–15
13. Chan, E., Lim, H.: Optimization and evaluation of shortest path queries. The VLDB Journal **16** (2007) 343–369
14. Pal, A., Paul, A., Mukherjee, A., Naskar, M., Nasipuri, M.: Fault detection and localization scheme for multiple failures in optical network. Distributed Computing and Networking (2008) 464–470