Finding k most Influential Edges on Flow Graphs

Petrie Wong[**], Cliz Sun[*], Eric Lo[*], Man Lung Yiu[*], Xiaowei Wu[**], Zhichao Zhao[**], T-H. Hubert Chan[**], Ben Kao[**]

[*] *Department of Computing, Hong Kong Polytechnic University*

[**] *Department of Computer Science, University of Hong Kong*

Abstract

In this paper, we formulate a novel question on maximum flow queries. Specifically, this problem aims to find which k edges would have the largest impact on a maximum flow query on a network. This problem has important applications in areas like social network and network planning. We show the inapproximability of the problems and present our heuristic algorithms. Experimental evaluations are carried out on real datasets and results show that our algorithms are scalable and return high quality solutions.

Keywords: Graph

1. Introduction

A flow network (or a flow graph) is a directed graph where each edge has a non-negative capacity and each edge receives a flow that does not exceed its capacity. The maximum flow problem, which identifies the maximum flow from a source vertex to a sink vertex in a flow network, has applications in web community detection [7, 10, 11], link spam detection [23], social community detection [14], online content voting [28], and network planning [18].

In this paper we study the problem of finding k edges that are the most influential to the maximum flow in a flow network G. Specifically, given a flow network $G = (V, E)$

[*]{csxzsun, ericlo, csmlyiu}@comp.polyu.edu.hk
[**]{kfwong2, xwwu, zczhao, hubert, kao}@cs.hku.hk

where V is the vertex set and E is the edge set with integer capacity, let F be the maximum flow from a source s to a sink t, we formulate two questions:

- The k Most Beneficial New Edges (kMBNE) — Given an edge set P s.t. $P \cap E = \emptyset$, which k edges in P would *maximize* F (if they are *inserted* into G)?
- The k Most Lethal Existing Edges (kMLEE) — Given an edge set $\overline{P} \subseteq E$, which k edges in $\overline{P}$ would *minimize* F (if they are *removed* from G)?

These two questions are encountered in many decision-support applications, some of which are discussed below.

Example 1. Social Network Marketing. Social networks, such as Facebook, are a popular platform for product marketing. Activities done by a user, e.g., *liking* a product page, are made known to his/her friends. If some of these friends follow the action (and *like* the product page), the product page is further made known to those friends' friends. We can represent a social network as a graph with users as vertices and friendships as edges. The amount of marketing information (such as the number of advertisement displayed to a user, or *impressions* from online advertising jargon) that passes on from one user a to a friend b can be modeled as the edge (a,b)'s capacity. The network flow out of a vertex set $S \subseteq V$ into a vertex set $T \subseteq V$ can be considered as the amount of marketing information that percolates through the network from the users in S to the users in T. If S is a set of users who are known to be interested in certain type of products, say, tablets, while T is a set of users who are potentially interested in those (e.g., T is a set of smartphone users), then a large max-flow from S to T would percolate more effectively the marketing effort done by a product manufacturer from S to T. This facilitates cross-promotion and makes the social network a more effective marketing platform.

One way to improve the flow from S to T is to add more edges (friendships) in the network. This can be done by friend recommendation. Some social network (e.g., Facebook) employs its own algorithm to produce a complete set of friend recommendations, denoted by P. Typically, the number of friends to be recommended to a user is in the hundreds, among which only a couple are displayed on the user's screen. An interesting question is which recommendations should the system pick and display? What

if the objective is to improve the network flow from a user set S to another set T?[1] The friend recommendation selection problem is an example of the kMBNE problem.

Example 2. Network Planning. A computer network is often modeled as a flow network where vertices are routers, edges are links between two routers, and edge capacities are the links' bandwidths. Recent works on network planning (e.g., [18]) focus on localizing faulty links *after* they break. In practice, it is also important to identify the set of the most risky links that will have the largest negative impact on the network throughput if they are broken. In this application, we may configure the set $\overline{P}$ to the entire set of edges E. This requires solving the kMLEE problem and the solutions help derive effective preventive measures (e.g., more frequent inspections on high-risk links) *before* any link break.

This paper investigates the kMBNE and kMLEE questions in the context of *maximum flow queries*. Our contributions are:

- First, we prove that kMBNE and kMLEE are ***inapproximable***. It is hard to find even an approximate solution (with *constant* approximation ratio), let alone find the exact solution.
- For both kMBNE and kMLEE, we develop polynomial-time heuristic algorithms that give high-quality solutions on real flow graphs. Moreover, we propose several pruning and optimization techniques to speedup our proposed algorithms.

The rest of the paper is organized as follows. We present our inapproximability results and heuristic solutions for the kMBNE and kMLEE problems in Sections 2 and 3, respectively. Section 4 presents the experiment results. Section 5 presents the related work. Section 6 concludes the work.

2. The kMBNE Problem

We first study the kMBNE problem. Given a flow graph $G = (V, E)$ and a candidate edge set P (with capacities) s.t. $P \cap E = \emptyset$, the problem is to *maximize* the flow

[1] Multiple sources and sinks there can be captured by a flow network with a supersource s' and supersink t', by connecting s' to each vertex in S, and each vertex in T to t', with infinite-capacity edges.

increment from a source vertex s to a sink vertex t in G by choosing k edges in P and inserting them into G. Note that P could be as large as *all* non-adjacent vertex-pairs in G if it is not explicitly given. We call the edges in P *new edges* to distinguish them from those already in G.

In this paper, we assume that edges have integer capacity $\|e_i\|$, for the sake of our hardness proofs. Nevertheless, our proposed algorithms are applicable to edges with non-integer capacity as well.

Let $F(G)$ be the maximum flow of (s,t) in graph G, and let $G^{e_i} = (V, E \cup \{e_i\})$ be the graph with $e_i(u_i, v_i) \in P$ inserted. The **benefit of** e_i, denoted by $B(e_i)$, is the increase in the maximum flow of (s,t) in G^{e_i}:

$$B(e_i) = F(G^{e_i}) - F(G). \tag{1}$$

The total **benefit of a k-edge set** $K \subseteq P$ is defined as:

$$B(K) = F(G^K) - F(G), \tag{2}$$

where G^K is the graph with all the edges in K inserted into G.

PROBLEM 1 (kMBNE). *Given a flow graph $G = (V, E)$ with integer capacity, an edge set P (with capacities) s.t. $P \cap E = \emptyset$, a source vertex s, and a sink vertex t; find a subset $K \subseteq P$ of k new edges that gives the largest total benefit i.e.,* $\arg\max_{K \subseteq P, |K|=k} B(K)$.

Note that in our basic definitions of kMBNE, increasing the network flow (F) is the sole objective. If we consider different edges to bear different costs (e.g., the running cost of a network link), then the flow network should be modeled as a weighted graph with each edge be given a weight that models a cost. If we define the *benefit* of a k-edge set $K \subseteq P$ to be a function of two factors: (1) the flow increment obtained by adding the edges in K to the network and (2) the total cost of the edges in K, then a variation of the kMBNE problem would be to find the k-edge set that maximizes the cost-accounted benefit. We remark that our algorithm can easily be extended to handle this variation as long as a cost-accounted benefit function is defined. For simplicity, we assume edge costs (weights) are 0 and focus on flow increment as the definition of benefit.

We illustrate an example problem instance in Figure 1. It shows a flow graph G and a set P of new edges (as dotted lines). The capacity of each edge is indicated by a number next to the edge. The maximum flow value (from s to t) on G is: $F(G) = 1$. When $k = 2$, the optimal solution is the set $K = \{(j, d), (i, a)\}$. After inserting K into G, the maximum flow value becomes: $F(G^K) = 5$. Thus, the benefit of K is: $B(K) = F(G^K) - F(G) = 5 - 1 = 4$.

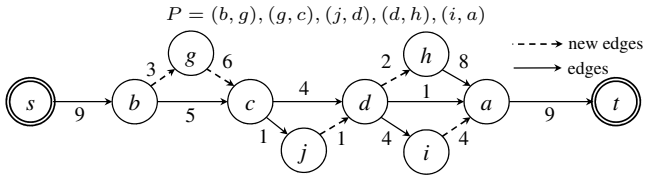


Figure 1: Problem instance of kMBNE

2.1. Problem Inapproximability

The kMBNE problem is a challenging problem. It is hard to find even an approximate solution, let alone find the exact solution. Specifically, Theorem 1 states that the kMBNE problem does not admit any polynomial-time algorithm with a constant approximation ratio, unless $\mathcal{NP}$-complete problems can be solved in quasi-polynomial time ZTIME$(n^{\text{polylog}(n)})$.

THEOREM 1. *There are no approximation algorithms for the kMBNE problem that run in polynomial-time and with constant approximation ratio, unless* $\mathcal{NP} \subseteq$ *ZTIME*$(n^{polylog(n)})$.

Proof: Observe that this theorem makes a stronger hardness assumption that NP is not contained in quasi-polynomial time ZTIME$(n^{polylog(n)})$, than the assumption that NP is not in polynomial time ZTIME(n^c). Hence, our reduction construction is allowed to take $O(n^{polylog(n)})$ time, instead of polynomial time.

First, we define a problem called MinEdge-MaxFlow. This problem aims at minimizing the size of the set K such that $B(K) = B(P)$; however, the size of K is not necessarily k.

PROBLEM 2 (MINEDGE-MAXFLOW). *Given a flow graph G and a set P of new edges with integer capacities, find a subset $K \subseteq P$ of minimum size such that $B(K) = B(P)$ (i.e., $F(G^K) = F(G^P)$).*

Next, we present the main idea of the proof:

- We first show in Lemma 1 that MinEdge-MaxFlow cannot be approximated within ratio $\log^{1+\epsilon} F(G^P)$ (unless $\mathcal{NP}$-complete problems can be solved in quasi-polynomial time).
- For the sake of contradiction, we assume that kMBNE can be approximated within a constant ratio c. With this assumption, we show in Lemma 2 that MinEdge-MaxFlow can be approximated within ratio $\mathcal{O}(\log F(G^P))$.
- Since the ratio $\mathcal{O}(\log F(G^P))$ (in Lemma 2) is smaller than the lower bound ratio $\log^{1+\epsilon} F(G^P)$ for MinEdge-MaxFlow (in Lemma 1), contradiction arises. Therefore, kMBNE cannot be approximated within a constant ratio c.

We next present Lemma 1 and Lemma 2 and their proofs.

LEMMA 1. *For any constant $\epsilon \in (0, 1)$, the MinEdge-MaxFlow problem cannot be approximated within ratio $\log^{1+\epsilon} F(G^P)$ in polynomial time, unless $\mathcal{NP} \subseteq$ ZTIME$(n^{polylog(n)})$.*

Proof 1. *We prove this lemma by reducing the Group-Steiner-Tree problem [9] to MinEdge-MaxFlow.*

The Group-Steiner-Tree problem (GST) [9] *Given an undirected graph $G = (V, E)$, a specified root vertex r, a collection of subsets (called* groups*) $g_1, \ldots, g_d \subseteq V$ and a length $w_e \geq 0$ for each edge $e \in E$, the problem is to construct a minimum-length sub-tree T^* (rooted at r) that spans at least one vertex from every group.*

Properties of GST [9] *For every constant $\epsilon \in (0, 1)$, the GST problem cannot be approximated in polynomial time within ratio $\log^{1+\epsilon} d$, where d is the number of groups, unless $\mathcal{NP} \subseteq$ ZTIME$(n^{polylog(n)})$. This holds even when the input graph is a tree T and all edge lengths are integers at most $|V|^{polylog(|V|)}$. Thus, we focus on such a subclass*

of GST instances in the remaining proof.

Reduction from GST to MinEdge-MaxFlow *Given an GST instance with input tree T on V_0 rooted at r, d groups $g_1, g_2, \ldots, g_d \subseteq V_0$ and an integer length $w_e \geq 0$ for each edge e in T, we construct a MinEdge-MaxFlow instance of size $O(|V_0|^{polylog(|V_0|)})$ in time $O(|V_0|^{polylog(|V_0|)})$ such that the maximum possible flow is $F(G^P) = d$. Moreover, any feasible solution with q edges for the MinEdge-MaxFlow instance implies a feasible solution with total length at most q for the GST instance and also vice versa. Having this reduction, the inapproximability of GST implies the inapproximability of MinEdge-MaxFlow.*

Next we give the reduction construction and prove its correctness.

First we change T to $T'(V', E')$ by the following operations: (a) direct all edges in T from parents to children; (b) replace each directed edge $e = (x, y)$ of length w_e with an edge-path *of length w_e, adding $w_e - 1$ new vertices. Note that T' contains $O(|V_0|^{polylog(|V_0|)})$ number of vertices and edges.*

Then for each group $g_i, i = 1, 2, \ldots, d$, we create a representative *vertex v_{g_i} and let $V_g = \{v_{g_1}, v_{g_2}, \ldots, v_{g_d}\}$ be the collection of all representatives. Define $E_g^1 = \{(v_{g_i}, t) | 1 \leq i \leq d\}$ and $E_g^2 = \{(x_{ij}, v_{g_i}) | 1 \leq i \leq d$ and $x_{ij} \in g_i\}$.*

Define the input graph $G(V, E)$ of the MinEdge-MaxFlow instance as follows.

- *Let $V = V' \cup V_g \cup \{s, t\}$ be the collection of vertices.*
- *Let $E = E_g^1 \cup E_g^2 \cup \{(s, r)\}$ be the collection of edges.*
- *Let $P = E'$ be the collection of potential new edges.*
- *Let the capacities be 1 for all edges in E_g^1 and $+\infty$ for all other edges (actually setting them d works, too).*

Note that $F(G^P) = d$ and $F(G) = 0$.

Notice that the construction time is $O(|V_0|^{polylog(|V_0|)})$.

Obviously, given a solution for the GST instance with total length q, we can construct a feasible solution for the MinEdge-MaxFlow instance with q edges by choosing the corresponding edges. Next we show that for any subset $Q \subseteq P$, if $F(G^Q) = F(G^P) = d$, then Q must contain a sub-tree of T with total length at

most $|Q|$ that spans at least one vertex from every group.

Observe that there is no point for choosing an edge without choosing all other edges in the same edge-path, or choosing an edge-path disconnected from r in Q (they are useless for increasing the max flow). Hence, we assume any feasible solution $Q \subseteq P$ is a sub-tree of T' with edge-paths (which also implies a sub-tree of T).

Since $F(G^Q) = F(G^P) = d$, we know that exactly 1 flow will pass each representative v_{g_i}. Hence, for each group g_i, at least one vertex $x_{ij} \in g_i$ is connected to r by a path in Q from r to x_{ij}, which means that Q is a tree that spans at least one vertex from every group. Hence, using the same solution Q (and replacing edge-paths in T' with edges in T), we get a sub-tree of T with total length $q = |Q|$.

Since the GST cannot be approximated in polynomial time within ratio $O(\log^{1+\epsilon} d)$, MinEdge-MaxFlow cannot be approximated in polynomial time within ratio $O(\log^{1+\epsilon} F(G^P))$, as $d = F(G^P)$.

LEMMA 2. *If kMBNE can be approximated within some constant ratio c, then MinEdge-MaxFlow can be approximated within ratio $O(\log F(G^P))$.*

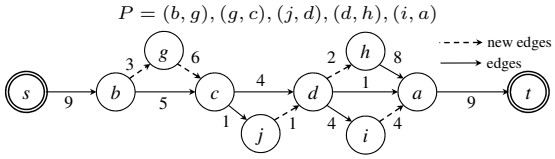
Proof 2. *Assume that a polynomial time algorithm $\mathcal{A}$ approximates kMBNE within some constant ratio c, then we design the following polynomial time algorithm $\mathcal{B}$ that approximates MinEdge-MaxFlow within ratio $O(\log F(G^P))$. This algorithm executes the following procedure for each k from 1 to $|P|$. For a fixed k, we run algorithm $\mathcal{A}$ on G iteratively for $N = \lceil \frac{\log F(G^P)}{\log \frac{1}{1-c}} + 1 \rceil$ times; after each iteration, we include the set of new edges to G and remove them from P. Let $K_k^{\cup}$ be the union of the sets of new edges obtained at a fixed k. Finally, we output the smallest union $K_k^{\cup}$ of new edges, among all values of k. Algorithm $\mathcal{B}$ runs in polynomial time because it calls $\mathcal{A}$ for $|P| \cdot N$ times.*

Notice that among all values of k, one of them, say k^*, is the optimal solution of MinEdge-MaxFlow. Fix that k^*, we know that the optimal solution of k^*MBNE is $F(G^P)$. Hence after running $\mathcal{A}$ on kMBNE for N times, the remaining target flow increase is at most $F(G^P)(1-c)^N \leq 1-c < 1$, which means that $F(G^{K_{k^*}^{\cup}}) = F(G^P)$ and algorithm $\mathcal{B}$ will output a solution with size at most $|K_{k^*}^{\cup}|$. Since k^* is the optimal solution for MinEdge-MaxFlow, we know that $|K_{k^*}^{\cup}| \leq N \cdot k^*$, which guarantees

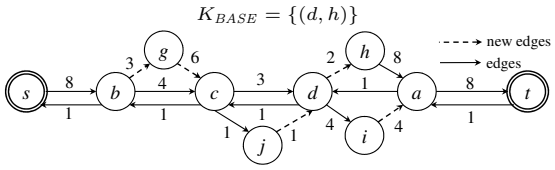
$N = O(\log F(G^P))$ *approximation ratio.*

2.2. Two-Phase Heuristics Algorithm (TPA)

Given the inapproximability result, it is highly likely that there does not exist a polynomial time approximation algorithm for solving the kMBNE problem with a *constant* approximation ratio. Therefore, our best hope in tackling the problem is to derive a heuristic algorithm with the following properties: (1) it runs in polynomial time, and (2) it scales well and returns high quality solutions in practice. In this section we describe our heuristics algorithm. We first start with a basic algorithm that satisfies property (1). Then, we add a heuristic in search of a high-quality answer and optimize the algorithm by a number of pruning techniques so that the optimized algorithm satisfies property (2).



(a) problem instance, (initial) flow graph G



(b) residual graph G w.r.t. the maximum flow

K_{GREEDY}={$(i, a), (j, d)$}

	Iteration 1		Iteration 2	
e_i	$F(G^{e_i})$	$B(e_i)$	$F(G^{e_i})$	$B(e_i)$
(b, g)	1	1-1=0	4	4-4=0
(d, h)	3	3-1=2	4	4-4=0
(g, c)	1	1-1=0	4	4-4=0
(i, a)	**4**	**4-1=3**	-	-
(j, d)	1	1-1=1	5	**5-4=1**

(c) running steps of TPA (k=2)

Figure 2: kMBNE Problem and TPA Running Steps

Given a flow graph G, we assume that the maximum flow $F(G)$ from a source s to a sink t has already been computed. Given G and $F(G)$, the *residual graph* G consists of edges that can admit non-zero flow [5][2]. Figure 2(a) shows an example flow graph G. The number printed next to an edge indicates the edge's capacity. The dotted edges are the new edges given by the edge-set P. In this example, the maximum flow is 1. Figure 2(b) shows the residual graph G w.r.t. the maximum flow[3]. The edges in G come in pairs: each edge e in G derives a backward edge (e.g., (c, b)), which carries the flow amount used for e, and a forward edge (e.g., (b, c)), which carries the residual capacity of e. The residual capacity is the original capacity of the edge minus the flow amount going through it in G. Any edges (forward or backward) with 0 capacity are removed from G. An *augmenting path* ρ is a path from s to t in the residual graph G [5]. If $F(G)$ is the maximum flow in G, then the residual graph w.r.t. $F(G)$ has no augmenting paths [5].

Our algorithm consists of two phases. In the first phase, our goal is to quickly find a base solution with a non-zero benefit. The second phase is to apply certain heuristics in search of a high-quality answer. The better of the two answers (from the two phases) is taken as the solution.

In the first phase, we insert *all* the new edges from P into the residual graph G. We then try to find a path ρ from s to t in G that uses the fewest number of the new edges[4]. If such a path exists, then ρ forms an augmenting path of the residual graph if those new edges in ρ had been inserted in P. This augmenting path brings at least an extra flow of 1 unit from s to t because edges assume integral capacities. If ρ contains not more than k new edges, then we can form a set K containing those new edges in ρ as an answer to the kMBNE problem. This results in a benefit of at least 1 unit (i.e., non-zero benefit). On the other hand, if ρ contains more than k new edges from P, then we know that no benefit can be obtained by adding any set of k edges from P into

[2] The residual graph G is a by-product of a typical max-flow algorithm. If it is not available, we execute a max-flow algorithm on G to obtain it.

[3] Only the solid lines are considered parts of the residual graph. The dotted lines are those from the new edge set P.

[4] The path ρ can be found by regarding the lengths of all new edges as 1, and the lengths of all other edges as 0, and then find the shortest path from s to t using a modified Breadth-First-Search, which takes $O(|V| + |E|)$ time.

the flow graph G.

After finding the base solution in the first phase, the second phase applies greedy heuristics to look for an alternate (hopefully better) solution. Greedy heuristics are known as a very practical heuristic to approach inapproximable problems [9]. We use a helper function `CalBenefit`, which determines the edge $e^* \in P$ that gives the maximum benefit among all edges in P. Algorithm 1 presents a naive implementation of `CalBenefit`. The procedure goes through every edge $e_i \in P$ and computes the benefit $B(e_i)$ of e_i by invoking an incremental max-flow algorithm `IncMaxFlow` on the graph G^{e_i}. The edge e^* that gives the highest benefit is returned.

Algorithm 1 Naive-Benefit (G, s, t, P) *implements* `CalBenefit`

1: **for each** new edge e_i in P **do**
2: `IncMaxFlow`(s, t, G, e_i)
3: calculate the benefit $B(e_i)$ of e_i
4: **return** the new edge whose benefit is the highest

We call the above naive implementation of `CalBenefit` Naive-Benefit. It runs in polynomial-time $\mathcal{O}(|P|IMF(G))$. We will present more efficient implementations of `CalBenefit` in Section 2.2.2.

Algorithm 2 shows the pseudo-code of our two-phase heuristic algorithm (TPA). Phase one (Lines 1–5) finds a base solution K_{BASE}, which provides a non-zero benefit. Phase two (Lines 6–11) finds another solution K_{GREEDY} by greedy heuristic. In phase two, the algorithm iterates k times. In each iteration, it invokes `CalBenefit` to greedily select the new edge $e^* \in P$ that gives the highest benefit $B(e^*)$. The edge e^* is then removed from P and inserted into G. After the k iterations, k edges from P would have been selected to form the set K_{GREEDY}. Finally, the algorithm compares the benefit of K_{BASE} and K_{GREEDY}, and returns the better one as the result (Lines 12–15).

Figure 2 demonstrates TPA. Suppose that $k = 2$. In the first phase, TPA considers the residual graph G and adds to it all the new edges from P (Figure 2(b)). It then attempts to find a path from s to t that uses the smallest number of new edges. In the example, one such path ρ is $s \to b \to c \to d \to h \to a \to t$, which includes just one

Algorithm 2 TPA (G, s, t, P, k)

▷ *Phase 1: Find a base solution with non-zero benefit*

let G be the residual graph w.r.t. the maximum flow from s to t
insert all edges in P to G
find the shortest path ρ (from s to t) in G ▷ by a modified breadth-first-search
if ρ exists and it contains at most k new edges **then**
    K_{BASE} = the new edges in ρ ▷ base solution

▷ *Phase 2: Apply greedy heuristic to refine the solution*

set $K_{GREEDY} = \emptyset$
for $i = 1$ to k **do**
    $e^* = \texttt{CalBenefit}(G, s, t, P)$ ▷ e^* has the highest benefit $B(e^*)$
    remove e^* from P
    insert e^* into K_{GREEDY}
    insert e^* into G ▷ update G accordingly
if $B(K_{BASE}) > B(K_{GREEDY})$ **then**
    return K_{BASE}
else
    return K_{GREEDY}

new edge $(d, h) \in P$. As a result, $K_{BASE} = \{(d, h)\}$. The benefit $B(K_{BASE})$ is 2.

In the second phase, TPA iterates a for-loop $k = 2$ times. Figure 2(c) shows the running steps of these two iterations. In the first iteration, TPA first invokes the function `CalBenefit` to find the most beneficial new edge (i, a). The benefit of the edge is $B((i, a)) = F(G^{(i,a)}) - F(G) = 4 - 1 = 3$. The edge (i, a) is removed from P and is inserted into both K_{GREEDY} and G. In the second iteration, `CalBenefit` is invoked the second time to find the next best edge in P that improves the flow of $G^{(i,a)}$. This edge is (j, d) and so it is inserted into the answer set K_{GREEDY}. Note that the benefit of $K_{GREEDY} = \{(i, a), (j, d)\}$ is 4. Finally, TPA compares the benefits of K_{BASE} and K_{GREEDY} and returns $\{(i, a), (j, d)\}$, as the final result.

2.2.1. Algorithm Analysis

Phase 1 of TPA takes $\mathcal{O}(|V|+|E|)$ to find a shortest path using a modified Breadth-First-Search algorithm. Phase 2 of TPA calls `CalBenefit` k times. With the naive implementation of `CalBenefit` (i.e., Algorithm 1), Phase 2 takes $\mathcal{O}(k \cdot |P| \cdot IMF(G))$

time.

Thus, the worst-case time complexity of TPA is: $\mathcal{O}(k \cdot |P| \cdot IMF(G) + |V| + |E|)$.

2.2.2. Efficiency Optimizations

Recall that the objective of `CalBenefit` is to determine the best new edge in P that maximizes the benefit. To do so, the naive implementation (Algorithm 1) calls an incremental max-flow algorithm `IncMaxFlow` $|P|$ times. In this section we introduce techniques that can achieve the same goal much more efficiently.

(a) Pruning (New) Edges in P

LEMMA 3. *Let γ be the benefit of the best new edge found so far. Given a new edge $e_i \in P$, if its capacity $\|e_i\|$ is not greater than γ, then e_i can be pruned because it cannot be the best new edge.*

In other words, during the process of finding the best new edge in P, we need not execute `IncMaxFlow` for any new edge e_i whose capacity $\|e_i\|$ is not greater than the benefit of the best new edge found so far. With Lemma 3, our approach will process the new edges in P in descending order of their capacities. This is because with such a descending capacity order, once we encounter a new edge e_i such that $\|e_i\| \leq \gamma$, all subsequent new edges in P can be pruned.

We next introduce another trick to prune a candidate new edge e_i, if it cannot be pruned by Lemma 3.

LEMMA 4. *A new edge $e_i = (u_i, v_i)$ can be pruned if there is no path from the source s to u_i, or from v_i to the sink t, in the residual graph* G.

Pruning using Lemma 4 incurs a small cost: we need to check in the residual graph G, whether u_i is reachable from s, and whether t is reachable from v_i. Although a BFS is needed to perform this checking, the cost is significantly smaller than executing the more expensive `IncMaxFlow` algorithm.

(b) "Solving Two Small Problems is Faster Than Solving One Big Problem" State-of-the-art (incremental) max-flow related algorithms all have super-linear time com-

plexity. Therefore, solving two half-sized max-flow problems would generally be faster than solving a full-sized one. The following lemma states a useful property:

LEMMA 5. *Given a flow graph G and its maximum flow $F(G)$, let G be the residual graph w.r.t. the maximum flow $F(G)$. Let $e = (u, v)$ be a new edge in P and let $F_{(x,y)}(\mathsf{G})$ be the maximum flow from a vertex x to a vertex y in the residual graph G. The benefit $B(e)$ of e is equal to* $\min(F_{(s,u)}(\mathsf{G}), \|e\|, F_{(v,t)}(\mathsf{G}))$.

Lemma 5 allows us to calculate the benefit of e by calling `IncMaxFlow` on two small problem instances: once using s as the source and u as the sink, and another one using v as the source and t as the sink. This generally is faster than calling `IncMaxFlow` on one large problem instance, which uses s as the source and t as the sink.

Lastly, we remark that, although we often use augmenting path to explain some concepts in this paper, all our lemmas and techniques are not specific to augmenting-path based (incremental) maxflow algorithms. That is because the residual graph, where the augmenting paths are found, can be deduced solely from the flow, which can be computed by push-relabel based (incremental) maxflow algorithms, too.

Algorithm Fast-Benefit Putting things above all together, we name this efficient implementation of `CalBenefit` as Fast-Benefit (see Algorithm 3). TPA uses this as its `CalBenefit` implementation.

Algorithm 3 Fast-Benefit(G, s, t, P) *implements* `CalBenefit`

1: $\gamma = 0$ ▷ the best benefit found so far
2: sort new edges of P in descending order of capacity $\|e_i\|$
3: **for each** new edge $e_i = (u_i, v_i)$ of P **do**
4: **if** $\|e_i\| \leq \gamma$ **then**
5: **break** ▷ Lemma 3
6: **if** $\neg$(u_i reachable from s) or $\neg$(t reachable from v_i) in G **then**
7: **continue** ▷ Lemma 4
8: $F_{(s,u_i)}(\mathsf{G}) =$ `IncMaxFlow`$(s, u_i, \mathsf{G}, \{\})$
9: $F_{(v_i,t)}(\mathsf{G}) =$ `IncMaxFlow`$(v_i, t, \mathsf{G}, \{\})$
10: $B(e_i) = \min(F_{(s,u_i)}(\mathsf{G}), \|e_i\|, F_{(v_i,t)}(\mathsf{G}))$ ▷ Lemma 5
11: update γ

3. The kMLEE Problem

We next study the kMLEE problem, which aims at *minimizing* the *maximum flow* of an integer-capacity flow graph $G = (V, E)$ from a source vertex s to a sink vertex t by choosing k existing edges in a given set $\overline{P} \subseteq E$ and removing them from G. As before, we assume that the edges have integer capacities. Note that $\overline{P}$ could be equal to E if it is not explicitly given.

Although some edges, i.e., $E \backslash \overline{P}$, are not removable in our setting, we can show that the problem when all edges are removable is no easier.

Given an instance of the kMLEE problem, we can multiply all capacities of edges with a large integer, e.g., $\frac{k}{\epsilon}$, so that all edges have integer capacity at least $\frac{k}{\epsilon}$. Then for all $e \in E \backslash \overline{P}$ with capacity c_e, we replace it with c_e parallel edges with capacity 1, and make all edges removable. Then in the new instance, all edges are removable, while the total contribution (to the decrease of maximum flow) of any set of at most k parallel edges of capacity 1 is at most an ϵ fraction of the total decrease.

Hence for small enough ϵ (while guaranteeing that $\frac{1}{\epsilon}$ is polynomially bounded), the solutions and objectives of the two problems are arbitrarily close. For presentation convenience, from now on we only consider the case when non-removable edges are allowed.

Let $e_i = (u_i, v_i)$ be an existing edge in $\overline{P}$. Let $G_{e_i} = (V, E - \{e_i\})$ be the graph with e_i removed. (Note: here we put e_i as a **subscript** of G, i.e., G_{e_i}, to denote the *removal* of e_i from G, whereas in the previous section we used e_i as a **superscript** of G, i.e., G^{e_i} to denote the *addition* of e_i into G). The damage of disconnecting u_i and v_i by removing e_i (or simply the **damage of** e_i), denoted by $D(e_i)$ is defined as:

$$D(e_i) = F(G) - F(G_{e_i}) \tag{3}$$

Let G_K be the graph G with a set of edges K removed from it. The total **damage of a set** K **of existing edges** is defined as:

$$D(K) = F(G) - F(G_K) \tag{4}$$

PROBLEM 3 (kMLEE). *Given a flow graph $G = (V, E)$ with integer capacity, an*

edge set $\overline{P} \subseteq E$, a source vertex s, and a sink vertex t; find a subset $K \subseteq \overline{P}$ of k edges that gives the largest total damage, i.e., $\arg\max_{K \subseteq \overline{P}, |K|=k} D(K)$.

We illustrate an example problem instance in Figure 3. It shows a flow graph G and a set $\overline{P}$ of removable edges (as double-lines). The capacity of each edge is indicated by a number next to the edge. The maximum flow value (from s to t) on G is: $F(G) = 5$. When $k = 2$, the optimal solution is the set $K = \{(d, h), (i, a)\}$. After removing K from G, the maximum flow value becomes: $F(G_K) = 0$. Thus, the damage of K is: $D(K) = F(G) - F(G_K) = 5 - 0 = 5$.

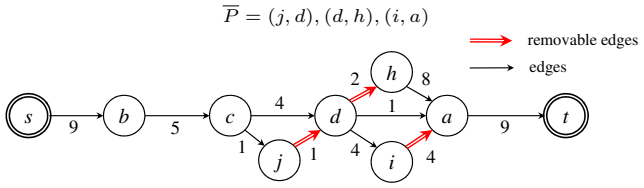


Figure 3: Problem instance of kMLEE

The kMLEE problem has been studied under the name *Network Interdiction Problem* [29, 22] and *Network Inhibition Problem* [20]. Wood [29] showed that the kMLEE problem is strongly $\mathcal{NP}$-hard, and gave a mixed integer programming formulation of the problem. However, no algorithm was given for solving the integer program. For efficient approximation algorithms, an FPTAS for the network inhibition problem is proposed in [20] for planar graphs.

THEOREM 2 ([29, 20]). *The kMLEE problem is strongly $\mathcal{NP}$-hard.*

In the following section, we strengthen the hardness results of the problem. In contrast to the problem on planar graphs [20], we show that unless $\mathcal{NP} = \mathcal{P}$, achieving an FPTAS for general flow networks is impossible. Indeed, we show that it is impossible to achieve any (non-zero) constant approximation ratio in polynomial time, unless $\mathcal{NP} = \mathcal{P}$.

3.1. Problem Inapproximability

THEOREM 3. *There is no polynomial time approximation algorithm for the kMLEE problem with non-zero constant approximation ratio, unless $\mathcal{NP} = \mathcal{P}$.*

Proof 3. *Consider the following simpler problem:*

PROBLEM 4. *Given a flow graph $G(V, E)$, a source vertex s, a target vertex t and a given set of edges $\overline{P} \subseteq E$, find a subset of k edges in $\overline{P}$ whose removal decreases the maximum flow from s to t by at least 1; or certify that it is impossible to do so.*

We show that the above problem is $\mathcal{NP}$-hard, which implies that it is $\mathcal{NP}$-hard to get any non-zero constant approximation ratio for the kMLEE problem in polynomial time, as otherwise the approximation algorithm can be used to solve the above problem in polynomial time.

Next we prove the $\mathcal{NP}$-hardness of Problem 4, by a reduction from the kMLEE problem (Theorem 2). For the sake of contradiction, assume that we have a polynomial-time algorithm $\mathcal{A}$ for Problem 4. Given an instance G of the kMLEE problem, we show that the problem can be solved by a polynomial number of calls of algorithm $\mathcal{A}$.

Recall that $F(G)$ is the maximum flow from s to t in G. For all $f = 1, 2, \ldots, F(G)$, we create the following instance of Problem 4 with $G'(V', E')$, s', t and the same set of removable edges $\overline{P}$.

Let $V' = V \cup \{s'\}$ and $E' = E \cup \{(s', s)\}$. Let the capacity of (s', s) be $F(G) - f + 1$. Note that the maximum flow from s' to t on G' is exactly $F(G) - f + 1$.

If there is a subset $X \subseteq \overline{P}$ of size k such that the maximum flow from s to t on $(V, E \backslash X)$ is at most $F - f$ (a solution for the kMLEE problem with objective f), then the maximum flow from s' to t on $(V', E' \backslash X)$ is decreased by at least 1.

If there is a subset $Y \subseteq \overline{P}$ of size k such that the maximum flow from s' to t on $(V', E' \backslash Y)$ is decreased by at least 1 (a solution for Problem 4), then the maximum flow from s to t on $(V, E \backslash Y)$ is at most $F - f$ (a solution for the kMLEE problem with objective at least f).

Hence we can solve the kMLEE problem by calling $\mathcal{A}$ on all $F(G)$ instances of Problem 4 and returning the solution with the maximum decrease in maximum flow, which contradicts Theorem 2.

Recall that we can reduce the kMLEE problem to the case when all edges are removable, i.e., $\overline{P} = E$, by multiplying all capacities by a large integer $\frac{k}{\epsilon}$, and splitting

non-removable edges. In this case, by changing the definition of Problem 4, i.e., to distinguish the case when we can decrease the maximum flow by at least $\frac{k}{\epsilon}$, and the case when it is impossible to decrease the maximum flow by more than k, we have the following immediate corollary of Theorem 3.

COROLLARY 1. *There is no polynomial-time approximation algorithm for the kMLEE problem when all edges are removable, i.e., $\overline{P} = E$, with approximation ratio $\Omega(\frac{1}{poly(|V|)})$, unless $\mathcal{NP} = \mathcal{P}$.*

3.2. Greedy Algorithm

Since we have proven that the kMLEE problem does not admit any polynomial time algorithm with non-zero approximation ratio unless $\mathcal{P} = \mathcal{NP}$, we directly devise a heuristic solution for the problem. In fact, the greedy heuristics employed in TPA for the kMBNE problem (see Section 2.2) can be applied here, resulting in a GREEDY algorithm for the the kMLEE problem.

Algorithm 4 is the pseudo-code of GREEDY. Specifically, it greedily chooses an edge $e \in \overline{P}$ with the largest damage, removes e from G, and repeats k times to form the set K. In each iteration, the edge with the largest damage can be identified by calling a function, `CalDamage`.

A naive implementation of `CalDamage` is to calculate the damage of **each** edge in $\overline{P}$ one-by-one, like the naive implementation of `CalBenefit` (Algorithm 1). We call this implementation Naive-Damage. It invokes `IncMaxFlow`[5] on graphs $\mathsf{G}_{e_1}, \cdots, \mathsf{G}_{e_{|\overline{P}|}}$, where G_{e_i} is the graph obtained by removing e_i from the residual graph G. We will present more efficient implementations for `CalDamage` in Section 3.2.2.

3.2.1. Algorithm Analysis

GREEDY calls an implementation of `CalDamage` k times. A naive implementation of `CalDamage` invokes an incremental max-flow algorithm $|\overline{P}|$ times. Thus, the worst-case time complexity of GREEDY is: $\mathcal{O}(k \cdot |\overline{P}| \cdot IMF(G))$.

[5] A typical *incremental* algorithm can also handle edge removal with the same complexity as the case of edge insertion.

Algorithm 4 kMLEE-Greedy (G, s, t, $\overline{P}$, k)

1: set $K_{GREEDY} = \emptyset$
2: **for** $i = 1$ to k **do**
3: $\quad e^* = \texttt{CalDamage}(G, s, t, \overline{P})$ ▷ e^* has the largest damage
4: $\quad$ remove e^* from $\overline{P}$, remove e^* from G
5: $\quad$ insert e^* into K_{GREEDY}
6: **return** K_{GREEDY}

3.2.2. Efficiency Optimizations

Recall that the objective of `CalDamage` is to determine the edge in $\overline{P}$ that has the largest damage for G. A naive implementation, Naive-Damage, calls an incremental algorithm $|\overline{P}|$ times. We introduce techniques to achieve the same goal much more efficiently.

(a) Pruning (Existing) Edges in $\overline{P}$

Let G be the residual graph w.r.t. the maximum flow $F(G)$, and let the graph G^* be the corresponding maximum flow network. Given an edge $e_i = (u_i, v_i)$, we define $f_{e_i}(G^*)$ as the flow value of e_i in G^*, which is equal to the residual capacity of the backward edge (v_i, u_i) in the residual graph G. For example, Figure 2(b), the backward edge (d, c) in G has a residual capacity of 1 unit, thus the flow through the edge (c, d) in the corresponding maximum flow network G^* is 1.

We use Lemma 6 to order and prune the edges in $\overline{P}$.

LEMMA 6. *Let γ be the damage of the edge in $\overline{P}$ that has the highest damage found so far. Given an edge $e_i \in \overline{P}$, if $f_{e_i}(G^*) \leq \gamma$, then e_i can be pruned because it cannot give the highest damage.*

(b) Damage Calculation Using a Shortcut in Incremental MaxFlow Algorithm

LEMMA 7. *Let $e = (u, v)$ be an edge in $\overline{P}$ and G_e be the graph obtained from removing the edge e (and its backward edge) from the residual graph G. The damage $D(e)$ of e is equal to* $\max(f_e(G^*) - F_{(u,v)}(\mathsf{G}_e), 0)$.

To explain the usage of Lemma 7, we first revisit the common two-step approach used in traditional incremental maxflow algorithms (both push-relabel-based

and augmenting-path-based) when handling the removal of an edge e_i: (Step 1) first push the existing flow via e_i back from the sink to the source and then remove e_i, to get an *intermediate* residual graph $\tilde{\mathsf{G}}_{e_i}$; (Step 2) resume maxflow calculation using $\tilde{\mathsf{G}}_{e_i}$. When that finishes, we obtain an *updated* residual graph G' and the updated maximum flow. Note that these two steps are necessary because for the incremental maxflow problem, we are expected to handle another edge insertion or edge deletion on G_{e_i} next. This makes the acquisition of G' for processing the next update necessary. In our context of computing `CalDamage`, there is a subtle difference with the above: we only care about the updated maxflow value, but we don't need G' because our next edge to be deleted, say, e_j, is not picked from G_{e_i}, but from the original input graph G. Therefore, Lemma 7 provides us a "shortcut" — we can skip Step 1 above and resume maximum flow calculation on G_e, using u as the source and v as the sink (a twist of Step 2).

Algorithm Fast-Damage Putting all the optimizations above together, we name our efficient implementation of `CalDamage` as Fast-Damage (see Algorithm 5). GREEDY uses this as its `CalDamage` implementation.

Algorithm 5 Fast-Damage(G, s, t, $\overline{P}$) *implements* `CalDamage`

1: $\gamma = 0$ ▷ the largest damage found so far
2: sort edges in $\overline{P}$ in descending order of their flow $f_{e_i}(G^*)$
3: **for each** edge $e_i = (u_i, v_i)$ of $\overline{P}$ **do**
4: **if** $f_{e_i}(G^*) \leq \gamma$ **then**
5: **break** ▷ Lemma 6
6: $F_{(u_i,v_i)}(\mathsf{G}_{e_i}) = \texttt{IncMaxFlow}(u_i, v_i, \mathsf{G}_{e_i}, \{\})$
7: $D(e_i) = \max(f_{e_i}(G^*) - F_{(u_i,v_i)}(\mathsf{G}_{e_i}), 0)$ ▷ Lemma 7
8: update γ

4. Experiments

In this section we present experiment results on real graphs. All experiments were conducted on a 2.5 GHz Intel PC running Ubuntu with 8 GB of RAM. We evaluated our algorithms TPA and GREEDY for the problems kMBNE and kMLEE, respectively. We measured the solution quality and the running time of our methods.

Our experiments use real directed graphs of different sizes and types. These datasets are listed in Figure 4. The source and the sink are selected randomly and the capacities are given in the datasets LINK and WAS or we randomly assign capacities $[1..10000]$ for edges in other datasets.[6] To simulate the kMBNE problem, we randomly pick an edge set P from existing edges of the graph. Then, we remove those edges from the graph and make the resulting graph the input to the kMBNE problem with P representing the set of new edges. For the kMLEE problem, the edge set $\overline{P}$ is selected randomly from existing edges of the graph.

Directed Graphs	Vertex #	Edge #	Avg. Degree
Washington flow network (WAS) http://dimacs.rutgers.edu/Challenges	131,074	392,960	5.57
CAIDA internet router topology (LINK) http://www.caida.org/tools/measurement/skitter	190,914	607,609	6.36
Stanford.edu web graph (SFW) http://snap.stanford.edu/data/	281,903	2,312,497	16.40
Epinions.com social network (ESN) http://snap.stanford.edu/data/	75,879	508,837	13.41
Pokec online social network (POK) http://snap.stanford.edu/data/	1,632,803	30,622,564	37.50
LiveJournal online social network (LIVE) http://snap.stanford.edu/data/	4,847,571	68,993,773	28.46

Figure 4: Real Graph Data Sets

4.1. The kMBNE Problem

In order to evaluate the solution quality of our TPA algorithm, we have implemented an impractical brute-force algorithm (BF) that exhaustively tries all combina-

[6]Experiments with capacities $[1..1000]$, $[1..100000]$, and $[1..1000000]$ are also conducted and the results are largely similar.

tions to find the optimal solution. The complexity of BF is $\mathcal{O}(\binom{|P|}{k} IMF(G))$, where $IMF(G)$ denotes the time complexity of an incremental maximum flow algorithm. We then measured the solution quality of TPA as the *actual approximation ratio* (A.A.R.), i.e., the solution benefit returned by TPA over the optimal benefit computed by BF. For comparison, we also implemented a Monte-Carlo heuristic algorithm, MCx, that randomly selects x edge-sets (each of size k) from P, evaluates the benefit of each edge-set, and returns the one with the highest benefit. MCx yields the best (1/x)·100% solution out of all possible edge-sets. Since our optimization techniques (Lemmas 3 and 4) are quite general, we also optimize MCx using them for fairness.

Initially, we first set $P = 25$ so that BF can compute the optimal solution in reasonable time. Figure 5 shows the solution quality (A.A.R.) of the methods on four real graphs from k=1 to 4.[7] Only A.A.R up to k=4 can be reported because BF cannot compute the optimal within weeks when $k \geq 5$. For the sake of comparison, we include the method TPA-BASE, which represents the first phase of TPA (see Algorithm 2).

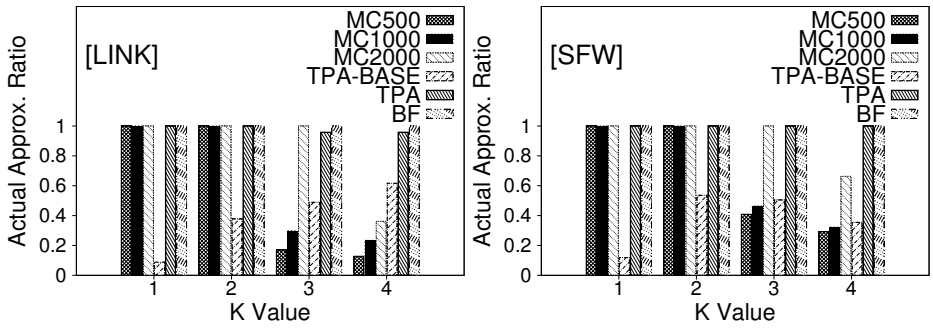
The A.A.R. of TPA is consistently equal to or close to 1.0 on all graphs on all k values. In contrast, MCx achieves an A.A.R of 1.0 only when $k \leq 2$, but that is simply because MCx has exhausted all cases as BF does. When $k \geq 3$, while MC2000 could still barely cover all cases as BF does, but MC500 and MC1000 could no longer cover all cases, and sharp drops in A.A.R are observed. As BF cannot compute optimal solutions (and thus A.A.R.) within feasible time when $k > 4$, so we directly compare the benefit of TPA and MCx after $k > 4$ and find that TPA consistently returns benefit at least 20 times better than MC2000 in all cases.

Figure 6 plots the running times of BF, TPA-BASE, TPA, and MCx for $k = 1$ to 20. We discard BF when $k \geq 5$ because it takes weeks. Even with our optimization techniques, MCx may be slower than BF (e.g., $k = 1$) because the value x there is larger than all possible cases—it manifests the difficulty of setting a practical x value for MCx.

From Figures 5 and 6, we see that TPA is clearly better than MCx because its solu-

[7] The conclusions obtained from datasets WAS and ESN are similar to the conclusions obtained from datasets LINK, SFW, POK and LIVE.

tions are consistently close to the optimal, more efficient, and get rid of the parameter x. TPA is slightly more expensive than TPA-BASE because TPA executes the second phase to refine the solution quality (see Algorithm 2). This additional computation is well paid off because TPA gives a close-to-optimal solution quality.

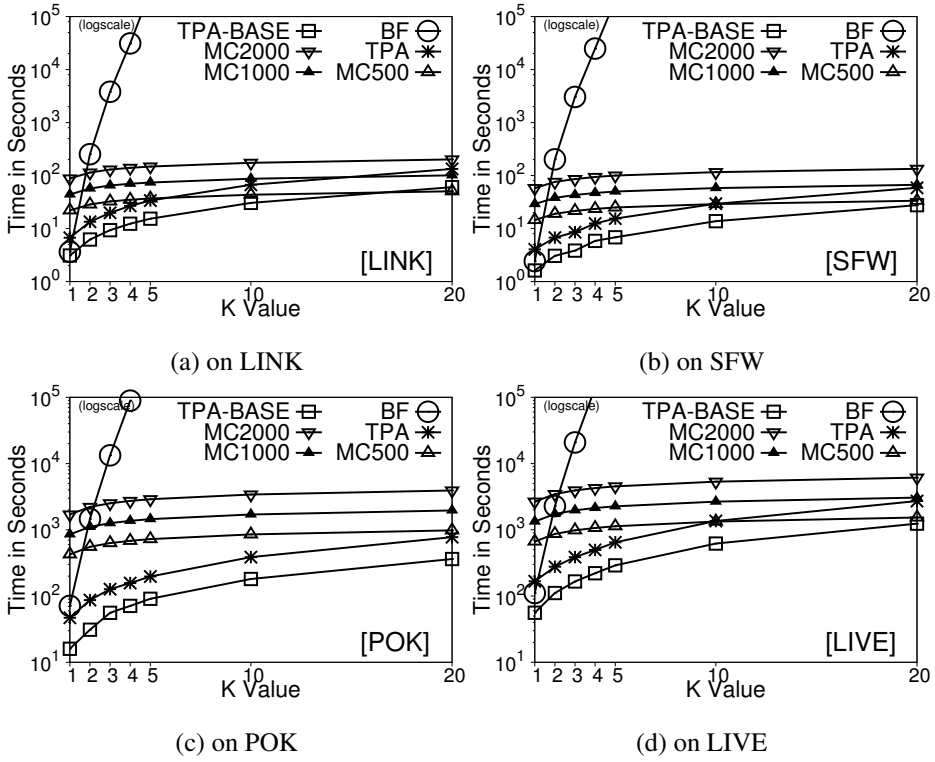


(a) on LINK (b) on SFW

(c) on POK (d) on LIVE

Figure 5: Quality vs. k, fixing $|P| = 25$ [for kMBNE problem]

As the performance of TPA is mainly k times the execution time of `CalBenefit` (Lines 8–9, Algorithm 2), we thus evaluate the scalability of two implementations of `CalBenefit`: Naive-Benefit (Algorithm 1) and Fast-Benefit (Algorithm 3). Figure 7 shows the scalability of these two implementations on four real graphs, by varying $|P|$ from 50 to 5000. Fast-Benefit is much more efficient than Naive-Benefit. The optimizations (in Fast-Benefit) are very effective so that many edges are pruned. Thus, Fast-Benefit can often stop early, and its running time is insensitive to the number of edges.



(a) on LINK (b) on SFW

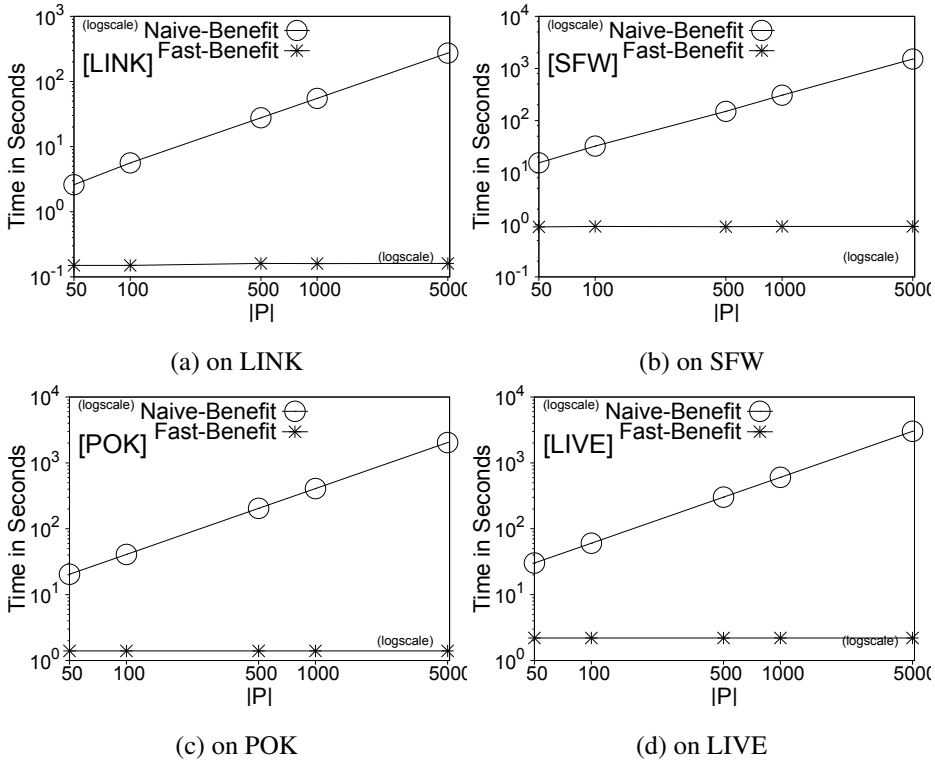
(c) on POK (d) on LIVE

Figure 6: Time vs. k, fixing $|P| = 25$ [for kMBNE problem]

Figure 8 (left) shows the scalability of Naive-Benefit and Fast-Benefit on all six real graphs ($|P| = 500$). Since real graphs are different in size and density, we first executed a maxflow algorithm (with random sources and sinks) on these graphs to measure the execution time and repeat the experiments many times to get an average. Then, we place those real graphs on the x-axis according to the average execution time. Figure 8 (left) shows that both implementations are scalable to graphs of different sizes and densities but Fast-Benefit outperforms Naive-Benefit by more than two orders of magnitude.

4.2. The kMLEE Problem

We next study the solution quality and the performance of the algorithms for the kMLEE problem. For this problem, our algorithm is GREEDY (Algorithm 4) and



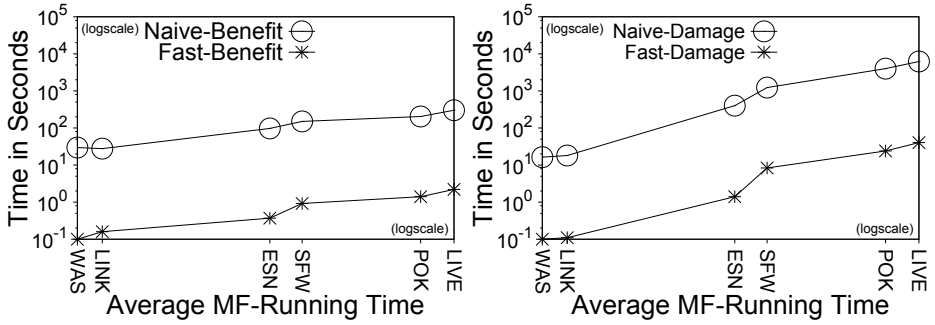
(a) on LINK (b) on SFW

(c) on POK (d) on LIVE

Figure 7: Time vs. $|P|$ [for kMBNE problem]

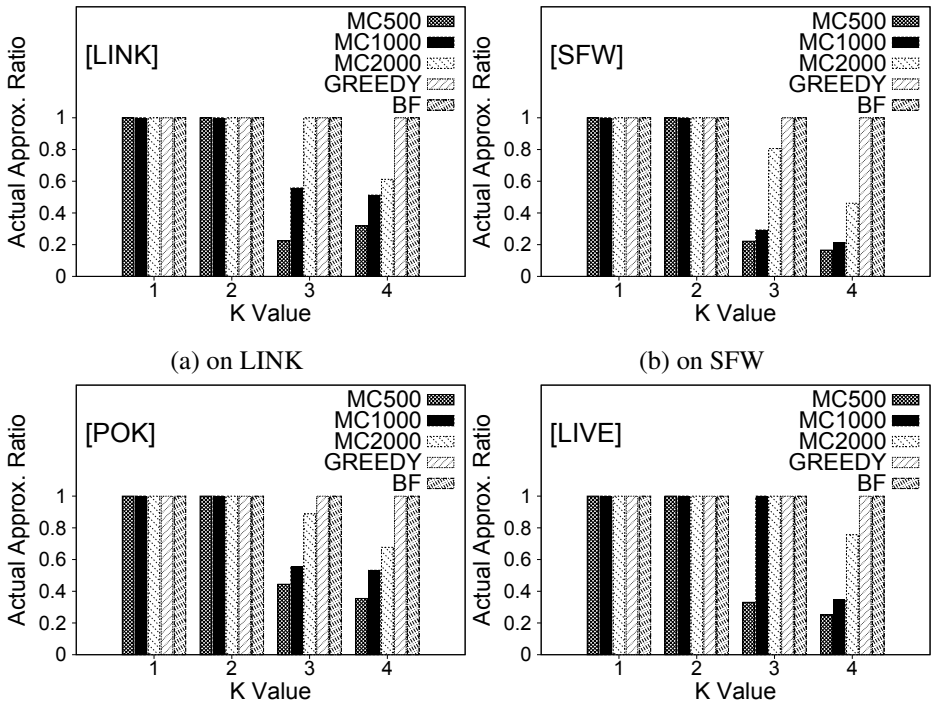
we again implemented a brute-force optimal algorithm BF. The complexity of BF is $\mathcal{O}(\binom{|\overline{P}|}{k} IMF(G))$, where $IMF(G)$ denotes the time complexity of an incremental maximum flow algorithm. We also implemented a Monte-Carlo algorithm, MCx, that applies our optimization (Lemma 6), for comparison.

We measure the solution quality of GREEDY as the *actual approximation ratio* (A.A.R.), i.e., the solution damage returned by GREEDY over the optimal damage computed by BF. Initially, we again first set $|\overline{P}| = 25$ so that BF can report the optimal solution in reasonable time. Figure 9 shows the solution quality of BF, GREEDY, and MCx on four real graphs. Again, only A.A.R up to $k = 4$ can be reported because BF cannot compute the optimal within weeks when $k \geq 5$. The kMLEE problem is so hard that there are no approximation algorithms with non-zero approximation ratio (see Theorem 3). Nevertheless, GREEDY achieves an A.A.R of 1.0 or close to 1.0 in all



(a) for kMBNE problem (b) for kMLEE problem

Figure 8: Scalability of `CalBenefit` and `CalDamage`



(a) on LINK (b) on SFW

(c) on POK (d) on LIVE

Figure 9: Quality vs. k, fixing $|\overline{P}| = 25$ [for kMLEE problem]

cases. In contrast, MCx shows sharp drop of quality once k gets larger. As BF cannot compute optimal solutions (and thus A.A.R.) within feasible time when $k > 4$, we

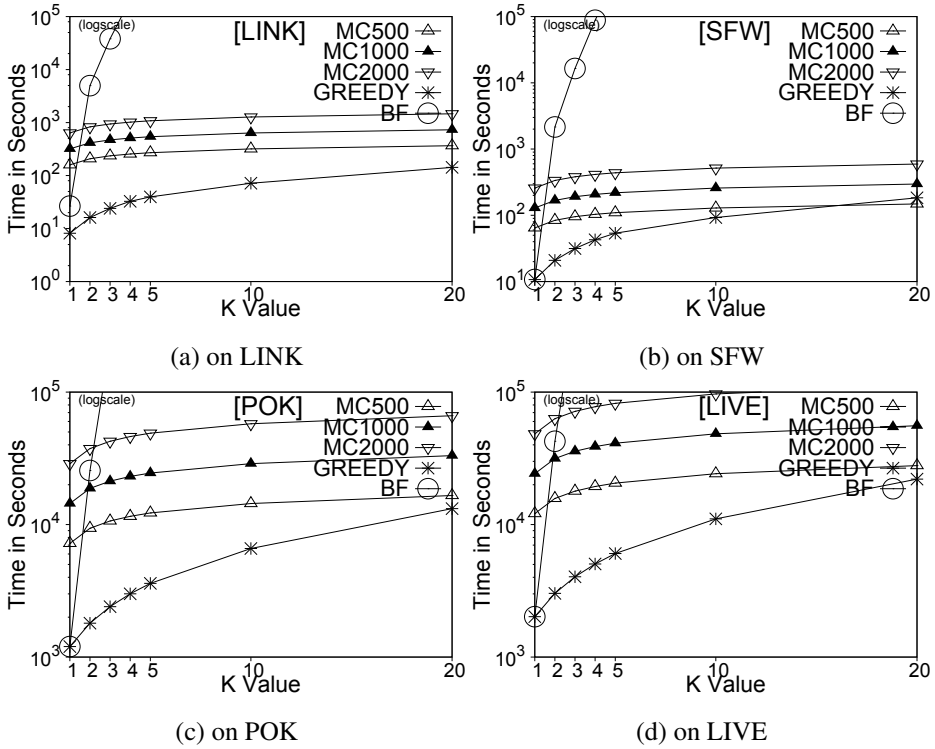
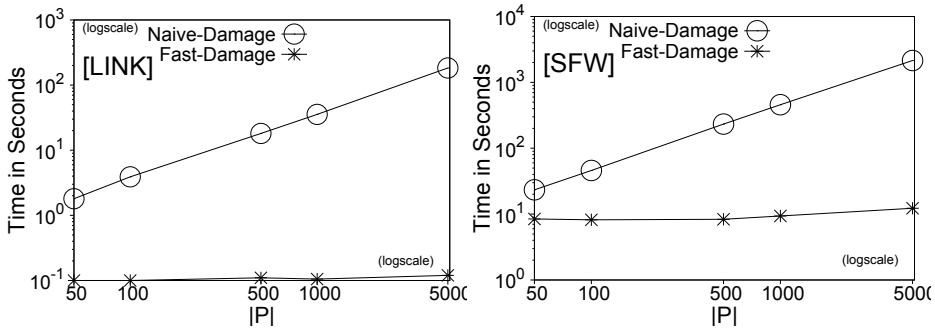


Figure 10: Time vs. k, fixing $|\overline{P}| = 25$ [for kMLEE problem]

directly compare the benefit of GREEDY and MCx after $k > 4$ and find that GREEDY consistently returns benefit at least 17 times better than MC2000 in all cases.

Figure 10 shows the running times of the methods for $k = 1$ to 20. We discard BF when $k \geq 5$ because it takes weeks. We observe that GREEDY clearly outperforms MCx in all cases. So, we conclude that GREEDY is clearly better than MCx for this kMLEE problem because its solutions are consistently close to the optimal, more efficient, and get rid of the parameter x.

As the performance of GREEDY is mainly k times the execution time of `CalDamage` (Lines 3–4; Algorithm 4), we thus evaluate the scalability of the two implementations of `CalDamage`: Naive-Damage and Fast-Damage (Algorithm 5). Figure 11 shows the scalability of these two implementations, by varying $|\overline{P}|$ from 50 to 5000. Fast-Damage is much more efficient than Naive-Damage. Due to our opti-



(a) on LINK (b) on SFW

(c) on POK (d) on LIVE

Figure 11: Time vs. $|\overline{P}|$ [for kMLEE problem]

mization technique, Fast-Damage is not very sensitive to the number of edges in the candidate set P. It is thus very scalable w.r.t. P. Figure 8 (right) shows the scalability of Naive-Damage and Fast-Damage on all six real graphs ($|\overline{P}| = 500$). The results show that both implementations are scalable to graphs of different sizes and densities. We see that Fast-Damage outperforms Naive-Damage in all cases.

5. Related Work

The broad applications of maximum flow make our work relevant to a number of areas:

Spatial Database *Optimal-location problems* (e.g., [8]) are a class of spatial decision-support problems where users look for the best location, l, out of a spatial extent, for

a new facility such that the greatest *benefit* is obtained. For example, [8] 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). Our work resembles optimal-location problems in terms of finding the edges (instead of points) to insert/remove from graph data (instead of spatial data) with the greatest benefit/damage.

Graph Processing There is a plethora of work on efficient graph processing, such as *sub-graph matching* (e.g., [32]), *super-graph matching* (e.g., [24]), and *graph similarity search* (e.g.. [34]). The focus of these works is to reduce the number of calls to some expensive procedures (e.g., graph isomorphism tests) in order to speed up the processing time. In this respect, our proposed optimizations have similar flavor as those because we also aim to reduce the number of calls to the expensive maximum flow algorithm. GConnect [1] is a work for supporting connectivity queries on disk-resident flow graphs. It can complement the incremental maximum flow algorithm that we currently employed as a procedure when dealing with disk-resident graphs.

The studies of *dynamic graph maintenance* provide efficient algorithms to update a certain graph measure (e.g., maximum flow [12]) with respect to the updates on nodes and edges in the graph. However, dynamic graph maintenance problems do not tell users which set of graph elements is worthwhile to get updated, which is the objective of this work.

Graph augmentation problems find a set of edges to add to a graph to satisfy a specified property. For instance, Meyerson and Tagiku [17] consider the problem of finding k edges to add to the graph in such a way to minimize the average distance between the nodes. Laoutaris et al. [13] examine a game theory problem where multiple players compete for purchasing links and each player attempts to minimize her distances to other nodes. Demaine and Morteza [6] study the problem of adding k edges to minimize the diameter with the goal of speeding up communication in an existing network design. Papagelis et al. [19] add k edges to minimize the characteristic paths. Tong et al. [27] investigate how to influence the spread of information in a network by inserting or deleting k edges. These works are orthogonal to us because they have not

considered the network flow as the property.

The goal of *graph simplification* is to prune edges while preserving some properties. For instance, Toivonen et al. [26] as well as Zhou et al. [33], prune edges while keeping the quality of best paths between all pairs of nodes, where quality is defined on concepts such as shortest path or maximum flow. Mathioudakis et al. [16] sparsify a network while taking care of maintaining the information propagation properties of the network. However, these works are different from our problem and they have not derived corresponding pruning techniques like us.

Finally, the *network inhibition* problem (NIP) [21] and its variant [2] delete edges from flow graphs in order to minimize the network flow, such that the total deletion cost (of selected edges) is within a specified budget. Our kMLEE problem is different from NIP in two aspects: (i) the hardness for approximation, (ii) the types of solutions. First, there exist approximation algorithms for NIP [21, 3]. In contrast, for our kMLEE problem, it is hard to find a polynomial-time approximation algorithm with a non-zero approximation ratio (cf. Theorem 2). Such a proof cannot be found in [21, 3]. Second, kMLEE and NIP have different types of solutions. kMLEE requires making binary decisions on edges; either remove an edge completely or keep it. NIP allows making fractional decisions on edges; it is possible to remove a fraction α (where $0 \leq \alpha \leq 1$) of capacity of an edge. One might use a heuristic to convert a NIP solution into a kMLEE solution (e.g., keeping k edges with the largest reduction on capacity). However, it is challenging to prove that such a heuristic can always preserve the quality of the solution. Thus, we consider this issue beyond the scope of our manuscript.

Social Networks and Information Propagation Tian et al. [25] study the link revival problem, where the objective is to turn existing edges with a few interactions to be more active, so that the resulted connection will improve the social network connectivity. Chaoji et al. [4] and Li et al. [15] study the problem of recommending connections that boost content propagation in a social network without compromising on the relevance of the recommendations. These works generally focus more on how to model the information as the edge weights. This paper, however, focuses more on the problem

hardness and the efficient algorithms to approach the problem.

Operations Research Our work bears resemblance to two problems in operations research (OR) and we explain the differences as follows. In OR, 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. For example, an inverse maximum flow problem [30], takes a feasible flow, f, and outputs a set of edge weight adjustments that make f become the maximum flow. In *reverse optimization problems*, users input a target value v, 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 [31], 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 OR problems require the user to **explicitly input the target to be tuned** (e.g., a feasible maximum flow f or the desired shortest-path distance d). In contrast, our work devises *scalable*, *efficient* and *practical* algorithms to **tell the user the target's identity as well as its optimized value**.

Network Planning Most network maintenance works (e.g., [18]) aim to localize faulty links *after* some links break. Our proposed kMLEE problem can be used to determine critical links in a network such that effective preventive maintenance measures can be implemented *before* any link breaks.

6. Conclusions

In this paper we study two interesting problems related to maximum flow F in a flow network $G = (V, E)$: the k Most Beneficial New Edges (kMBNE) problem and the k Most Lethal Existing Edges (kMLEE) problem. The two problems have applications including social network marketing and network planning. Our theoretical results show that the two problems are **inapproximable**. Thus, we devise polynomial-time heuristic algorithms to solve the kMBNE and kMLEE problems. We also devise optimization techniques to significantly speedup our algorithms. We evaluate our algorithms using real datasets that embrace the motivating applications. Experimental

results show that our algorithms run efficiently and yield high quality solutions.

References

[1] C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: a connectivity index for massive disk-resident graphs. *PVLDB*, 2(1):862–873, 2009.

[2] D. S. Altner, Ö. Ergun, and N. A. Uhan. The maximum flow network interdiction problem: Valid inequalities, integrality gaps, and approximability. *Oper. Res. Lett.*, 38(1):33–38, 2010.

[3] C. Burch, R. Carr, S. Krumke, M. Marathe, C. Phillips, and E. Sundberg. A decomposition-based pseudoapproximation algorithm for network flow inhibition. *Network Interdiction and Stochastic Integer Programming*, 22:51–68, 2003.

[4] V. Chaoji, S. Ranu, R. Rastogi, and R. Bhatt. Recommendations to boost content spread in social networks. In *WWW*, pages 529–538, 2012.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[6] E. D. Demaine and M. Zadimoghaddam. Minimizing the diameter of a network using shortcut edges. In *SWAT*, pages 420–431, 2010.

[7] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *KDD*, pages 150–160, 2000.

[8] Y. Gao, B. Zheng, G. Chen, and Q. Li. Optimal location selection query processing in spatial databases. *TKDE*, 21:1162–1177, 2009.

[9] E. Halperin and R. Krauthgamer. Polylogarithmic inapproximability. In *STOC*, pages 585–594, 2003.

[10] N. Imafuji and M. Kitsuregawa. Effects of maximum flow algorithm on identifying web community. In *WIDM*, pages 43–48, 2002.

[11] N. Imafuji and M. Kitsuregawa. Finding a web community by maximum flow algorithm with hits score based capacity. In *DASFAA*, pages 101–106, 2003.

[12] S. Kumar and P. Gupta. An incremental algorithm for the maximum flow problem. *Journal of Mathematical Modeling and Algorithms*, 2:1–16, 2003.

[13] N. Laoutaris, L. J. Poplawski, R. Rajaraman, R. Sundaram, and S.-H. Teng. Bounded budget connection (bbc) games or how to make friends and influence people, on a budget. In *PODC*, pages 165–174, 2008.

[14] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008.

[15] D. Li, Z. Xu, S. Li, X. Sun, A. Gupta, and K. P. Sycara. Link recommendation for promoting information diffusion in social networks. In *WWW*, pages 185–186, 2013.

[16] M. Mathioudakis, F. Bonchi, C. Castillo, A. Gionis, and A. Ukkonen. Sparsification of influence networks. In *KDD*, pages 529–537, 2011.

[17] A. Meyerson and B. Tagiku. Minimizing average shortest path distances via shortcut edge addition. In *APPROX-RANDOM*, pages 272–285, 2009.

[18] A. Pal, A. Paul, A. Mukherjee, M. Naskar, and M. Nasipuri. Fault detection and localization scheme for multiple failures in optical network. In *ICDCN*, pages 464–470, 2008.

[19] M. Papagelis, F. Bonchi, and A. Gionis. Suggesting ghost edges for a smaller world. In *CIKM*, pages 2305–2308, 2011.

[20] C. A. Phillips. The network inhibition problem. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 776–785, 1993.

[21] C. A. Phillips. The network inhibition problem. In *STOC*, pages 776–785, 1993.

[22] J. O. Royset and R. K. Wood. Solving the bi-objective maximum-flow network-interdiction problem. *INFORMS Journal on Computing*, 19(2):175–184, 2007.

[23] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *AIRWeb*, 2007.

[24] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *ICDE*, pages 637–648, 2010.

[25] Y. Tian, Q. He, Q. Zhao, X. Liu, and W.-c. Lee. Boosting social network connectivity with link revival. In *CIKM*, pages 589–598, 2010.

[26] H. Toivonen, S. Mahler, and F. Zhou. A framework for path-oriented network simplification. In *IDA*, pages 220–231, 2010.

[27] H. Tong, B. A. Prakash, T. Eliassi-Rad, M. Faloutsos, and C. Faloutsos. Gelling, and melting, large graphs by edge manipulation. In *CIKM*, pages 245–254, 2012.

[28] D. N. Tran, B. Min, J. Li, and L. Subramanian. Sybil-resilient online content voting. In *NSDI*, pages 15–28, 2009.

[29] R. K. Wood. Deterministic network interdiction. *Mathematical and Computer Modelling*, 17(2):1–18, 1993.

[30] C. Yang and J. Zhang. Inverse maximum flow and minimum cut problems. *Optimization*, 40:147–170, 1997.

[31] J. Zhang and Y. Lin. Computation of reverse shortest-path problem. *Journal of Global Optimization*, 25:243–261, 2003.

[32] S. Zhang, J. Yang, and W. Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1):1185–1194, 2010.

[33] F. Zhou, S. Mahler, and H. Toivonen. Network simplification with minimal loss of connectivity. In *ICDM*, pages 659–668, 2010.

[34] Y. Zhu, L. Qin, J. X. Yu, Y. Ke, and X. Lin. High efficiency and quality: large graphs matching. In *CIKM*, pages 1755–1764, 2011.