Attacking Shortest Paths by Cutting Edges

BENJAMIN A. MILLER and ZOHAIR SHAFI, Northeastern University, USA
WHEELER RUML, University of New Hampshire, USA
YEVGENIY VOROBEYCHIK, Washington University in St. Louis, USA
TINA ELIASSI-RAD, Northeastern University, USA
SCOTT ALFELD, Amherst College, USA

Identifying shortest paths between nodes in a network is a common graph analysis problem that is important for many applications involving routing of resources. An adversary that can manipulate the graph structure could alter traffic patterns to gain some benefit (e.g., make more money by directing traffic to a toll road). This article presents the *Force Path Cut* problem, in which an adversary removes edges from a graph to make a particular path the shortest between its terminal nodes. We prove that the optimization version of this problem is APX-hard but introduce PATHATTACK, a polynomial-time approximation algorithm that guarantees a solution within a logarithmic factor of the optimal value. In addition, we introduce the *Force Edge Cut* and *Force Node Cut* problems, in which the adversary targets a particular edge or node, respectively, rather than an entire path. We derive a nonconvex optimization formulation for these problems and derive a heuristic algorithm that uses PATHATTACK as a subroutine. We demonstrate all of these algorithms on a diverse set of real and synthetic networks, illustrating where the proposed algorithms provide the greatest improvement over baseline methods.

CCS Concepts: • **Theory of computation** → **Shortest paths**; *Stochastic approximation*; *Network optimization*; • **Security and privacy** → *Information flow control;*

Additional Key Words and Phrases: Adversarial graph analysis, APX-hardness, integer programming

ACM Reference format:
Benjamin A. Miller, Zohair Shafi, Wheeler Ruml, Yevgeniy Vorobeychik, Tina Eliassi-Rad, and Scott Alfeld. 2023. Attacking Shortest Paths by Cutting Edges. *ACM Trans. Knowl. Discov. Data.* 18, 2, Article 35 (November 2023), 42 pages.
https://doi.org/10.1145/3622941

BAM was supported by the United States Air Force under Contract No. FA8702-15-D-0001. TER was supported in part by the Combat Capabilities Development Command Army Research Laboratory (under Cooperative Agreement No. W911NF-13-2-0045) and by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. YV was supported by grants from the Army Research Office (W911NF1810208, W911NF1910241) and National Science Foundation (CAREER Award IIS-1905558). Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.
Authors' addresses: B. A. Miller, Z. Shafi, and T. Eliassi-Rad, Northeastern University, Boston, MA, 02115; e-mails: {miller.be, shafi.z, t.eliassirad}@northeastern.edu; W. Ruml, University of New Hampshire, Durham, NH, 03824; e-mail: ruml@cs.unh.edu; Y. Vorobeychik, Washington University in St. Louis, St. Louis, MO, 63130; e-mail: yvorobeychik@wustl.edu; S. Alfeld, Amherst College, Amherst, MA, 01002; e-mail: salfeld@amherst.edu.
Author's current address: B. A. Miller, MIT Lincoln Laboratory, Lexington, MA, 02421; e-mail: bamiller@ll.mit.edu.

1556-4681/2023/11-ART35 $15.00
https://doi.org/10.1145/3622941

1 INTRODUCTION

Finding the shortest paths between interconnected entities is an important task in a wide variety of applications. When routing resources—such as traffic on roads, ships among ports, or packets among routers—identifying the shortest path between two nodes is key to making efficient use of the network. Given that traffic prefers to take the shortest route, a malicious adversary with the ability to alter the graph topology could manipulate the paths to gain an advantage. For example, a road network could be manipulated to direct traffic between two popular locations across a toll road the adversary owns. A computer network could be altered to encourage packets to flow through an adversary's subnetwork. Undermining connections in a social network may enable the adversary to become a "gatekeeper" to an important individual. Countering such behavior is important, and understanding vulnerability to such manipulation is a step toward more robust graph mining.

In this article, we present the *Force Path Cut* problem, in which an adversary wants the shortest path between a source node and a target node in an edge-weighted network to follow a preferred path. The adversary achieves this goal by cutting edges, each of which has a known cost for removal. We show that the optimization version of this problem is APX-hard via a reduction from the 3-Terminal Cut problem [18]. To optimize Force Path Cut, we recast it as a Weighed Set Cover problem, which allows us to use well-established approximation algorithms to minimize the total edge removal cost. We propose the PATHATTACK algorithm, which combines these algorithms with a constraint generation method to efficiently identify a subset of paths to target for disruption. While these algorithms only guarantee an approximately optimal solution in general, PATHATTACK yields the lowest-cost solution in a large majority of our experiments.

We also introduce the *Force Edge Cut* and *Force Node Cut* problems, where a specific edge (or node) is targeted rather than an entire path. The optimization versions of these problems are also APX-hard, and we use PATHATTACK as part of a heuristic search algorithm to solve them. The three problems are defined formally in the following section.

1.1 Problem Statement

We consider a graph $G = (V, E)$, where the vertex set V contains N entities and E consists of M edges, which may be directed or undirected. Each edge has a weight $w : E \to \mathbb{R}_{\geq 0}$ denoting the expense of traversal (e.g., distance or time). In addition, each edge has a removal cost $c : E \to \mathbb{R}_{\geq 0}$. We are also given a source node $s \in V$, a target node $t \in V$, and a budget $b > 0$ for edge removal. Within this context, there are three problems we address:

- **Force Edge Cut:** Given an edge $e^* \in E$, find $E' \subset E$ where $\sum_{e \in E'} c(e) \leq b$ and all shortest paths from s to t in $G' = (V, E \setminus E')$ use e^*.
- **Force Node Cut:** Given a node $v^* \in V$, find $E' \subset E$ where $\sum_{e \in E'} c(e) \leq b$ and all shortest paths from s to t in $G' = (V, E \setminus E')$ use v^*.
- **Force Path Cut:** Given a path p^* from s to t in G, find $E' \subset E$ where $\sum_{e \in E'} c(e) \leq b$ and p^* is the unique shortest path from s to t in $G' = (V, E \setminus E')$.

Each variation of the problem addresses a different adversarial objective: there is a particular edge (e.g., a toll road), a particular node (e.g., a router in a network), or an entire path (e.g., a sequence of surveillance points) where increased traffic would benefit the adversary. The attack vector in all cases is removal of edges, and the adversary has access to the entire graph. We also consider the corresponding optimization problems for each decision problem defined above. In the optimization version, the objective is to find E' with the smallest cost $\sum_{e \in E'} c(e)$ that satisfies the decision problem's conditions. We refer to each optimization counterpart of the corresponding

decision problem as Optimal Force Edge Cut, Optimal Force Node Cut, and Optimal Force Path Cut, respectively.

1.2 Contributions

The main contributions of this article are as follows:

- We define the Force Edge Cut, Force Node Cut, and Force Path Cut problems and prove that their corresponding optimization problems are APX-hard.
- We introduce the PATHATTACK algorithm, which provides a logarithmic approximation for Optimal Force Path Cut with high probability (greater than $1 - (1/|E|)$) in polynomial time.
- We provide a non-convex optimization formulation for Optimal Force Edge Cut and Optimal Force Node Cut, as well as polynomial-time heuristic algorithms.
- We present the results of over 16,000 experiments on a variety of synthetic and real networks, demonstrating where these algorithms perform best with respect to baseline methods.

1.3 Article Organization

The remainder of this article is organized as follows. In Section 2, we briefly summarize related work on inverse optimization and adversarial graph analysis. In Section 3, we provide a sketch of the proof that all three optimization problems are APX-hard. (Details are presented in Appendix A.) Section 4 introduces the PATHATTACK algorithm, which guarantees a logarithmic approximation of the optimal solution to Force Path Cut. In Section 5, we show how PATHATTACK can be used as a heuristic for the optimization versions of Force Edge Cut and Force Node Cut. Section 6 documents the results of experiments on diverse real and synthetic networks, demonstrating where PATHATTACK provides a benefit over baseline methods. In Section 7, we conclude with a summary and discussion of open problems.

2 RELATED WORK

Early work on attacking networks focused on disconnecting them [2]. This work demonstrated that targeted removal of high-degree nodes was highly effective against networks with power law degree distributions (e.g., Barabási–Albert networks), but far less so against random networks. This is due to the prevalence of hubs in networks with such degree distributions. Other work has focused on disrupting shortest paths via edge removal, but in a narrower context than ours. Work on the most vital edge problem (e.g., [44]) attempts to efficiently find the single edge whose removal most increases the distance between two nodes. (The similarly defined most vital node seeks the node whose removal does the same [45].) Our present work, in contrast, considers a devious adversary that wishes a particular path to be shortest, or to target a specific node or edge for routing, rather than to maximize the distance.

There are several other adversarial contexts in which path finding is highly relevant. Some work is focused on traversing hostile territory, such as surreptitiously planning the path of an unmanned aerial vehicle [29]. The complement of this is network interdiction, where the goal is to intercept an adversary who is attempting to traverse the graph while remaining hidden. Bertsimas et al. formulate an optimization problem similar to Optimal Force Path Cut, where the goal is overall disruption of flows rather than targeting a specific shortest path [6]. Network interdiction has been studied in a game theoretic context for many years [52] and has expanded into work on disrupting attacks, with the graph representing an attack plan [36]. In that work, as in ours, oracles can be used to avoid enumerating an exponentially large number of possible strategies [26].

As with many graph problems, finding a shortest path can be formulated as an optimization problem: over all paths from s to t, find the one that minimizes the sum of edge weights. Finding the weights for two nodes to have a particular shortest path is an example of inverse optimization [1]. From this perspective, the shortest path is a function mapping edge weights to a path, and the inverse shortest path problem is to find the inverse function: input a path and output a set of weights. This typically involves finding a new set of weights given a baseline that should change as little as possible, with respect to some distance metric (though it is shown in [56] that solving the problem without baseline weights solves the minimum cut problem). To solve inverse shortest path while minimizing the L_1 norm between the weights, Zhang et al. propose a column generation algorithm [59], similar to the constraint generation method we describe in Section 4.3. Such a constraint generation procedure is also used in the context of navigation meshes in [9]. The instance of inverse shortest path that is closest to Force Path Cut uses the weighted Hamming distance between the weight vectors, where changing the weight of an edge has a fixed cost, regardless of the size of the change.[1] This case was previously shown to be NP-hard [57]. In this article, we show that Optimal Force Path Cut, which is less flexible, is also not just NP-hard, but APX-hard.

In addition to inverse shortest path, authors have considered the inverse shortest path length problem (sometimes called reverse shortest path) [47], where only the length of the shortest path is specified. This problem has been shown to be NP-hard except when only considering a single source and destination [16, 58]. There is also the notion of inverse shortest path routing, in which edge weights are manipulated so that edges from one specified subset are used for shortest paths, while edges from a second, disjoint subset are never used [10]. Other graph-based optimization problems, such as maximum flow and minimum cut, have also been topics in the inverse optimization literature (see, e.g., [20, 27, 38]). Also on the topic of altering a graph to change the routing pattern is work on altering node delays within a given budget [21, 39], including cases where the improvement must be noticeable to users [40]. Others have considered the problem of altering shortest paths by adding edges, rather than removing them or changing weights [41].

There has recently been a great deal of work on attacking machine learning methods where graphs are part of the input. Finding shortest paths is another common graph problem that attackers could exploit. Attacks against vertex classification [14, 19, 54, 55, 60] and node embeddings [8, 11] consider attackers that can manipulate edges, node attributes, or both in order to affect the outcome of the learning method. In addition, attacks against community detection have been proposed where a node can create new edges to alter its group assignment from a community detection algorithm [13, 28, 30], or to reduce the efficacy of community detection overall [12, 23, 37, 43, 51]. Our work complements these efforts, expanding the space of adversarial graph analysis into another important graph mining task.

3 COMPUTATIONAL COMPLEXITY

We first consider the complexity of Force Path Cut. Like inverse shortest path under Hamming distance [57], this problem is NP-hard, as shown in our prior work [42]. A novel result of our present work is that Optimal Force Path Cut is not merely NP-hard; there is no possible polynomial-time approximation scheme—i.e., no polynomial-time algorithm can find a solution within a factor of $(1 + \epsilon)$ of the optimal (minimal) budget unless P = NP. This applies to the optimization version of the problem. Using a linear reduction from another APX-hard problem, we prove the following theorem.

[1]Edge removal can be simulated by significantly increasing weights. Inverse shortest path using weighted Hamming distance also allows for reducing edge weights, which is not allowed in Force Path Cut.

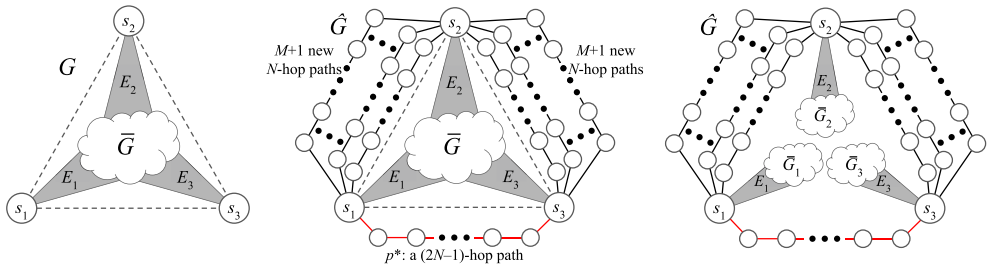

Fig. 1. Reduction from 3-Terminal Cut to Force Path Cut. The initial graph (left) includes three terminal nodes s_1, s_2, and s_3, which are connected to the rest of the graph by edge sets E_1, E_2, and E_3, respectively. The dashed lines indicate the possibility of edges between terminals. The input to Force Path Cut, $\hat{G}$ (center), includes the original graph plus many long parallel paths between terminals. A single path from s_1 to s_3 composing p^* is indicated in red. The result of solving Force Path Cut and removing the returned edges (right) is that any existing paths between the terminals have been removed, thus disconnecting them in the original graph and solving 3-Terminal Cut.

Theorem 3.1. *Optimal Force Path Cut is APX-hard, including the case where all weights and all costs are equal.*

3.1 Proof Sketch

The following is a sketch of the proof of Theorem 3.1, with a detailed proof provided in Appendix A. First, consider the case for undirected graphs, where all edges weights are equal to their removal costs. We prove this is APX-hard via reduction from 3-Terminal Cut [18]. In 3-Terminal Cut, we are given a graph $G = (V, E)$; three terminal nodes $s_1, s_2, s_3 \in V$; and a budget $b \geq 0$. For the purpose of this proof, we only consider the case where all edge weights are equal. The goal is to determine whether there exists $E' \subset E$, with $|E'| \leq b$, where s_1, s_2, and s_3 are disconnected in $G' = (V, E \setminus E')$; i.e., after the edges E' are removed, no path exists between any of the terminal nodes.

We reduce 3-Terminal Cut to Force Path Cut via the following process, illustrated in Figure 1. Create $M + 1$ new disjoint paths, each N hops long, from s_1 to s_2. This introduces $(M + 1)(N - 1)$ new nodes and $(M + 1)N$ new edges. Do the same for s_2 and s_3 (another $(M + 1)(N - 1)$ nodes and $(M + 1)N$ edges). Create a new $(2N - 1)$-hop path from s_1 to s_3 ($2N - 2$ new nodes, $2N - 2$ new edges). Make the new path from s_1 to s_3 the target path p^*, with $s = s_1$ and $t = s_3$. Call the resulting graph $\hat{G}$. Solve Force Path Cut on $\hat{G}$, obtaining a set of edges E' to be removed.

If E' is a solution to Force Path Cut, it is also a solution to 3-Terminal Cut. If any of the terminal nodes could be reached from another using edges in the original graph, p^* would not be the shortest path. If a path from s_2 to s_3 from the original graph remained, for example, this path would be at most $N - 1$ hops long. There would exist a path from s_1 to s_3 using one of the new $(N + 1)$-hop paths from s_1 to s_2, followed by the remaining path from s_2 to s_3. This path would be less than $2N + 1$ hops, so p^* would not be the shortest path.

In the optimization version of the problem, the goal is to find the E' that solves 3-Terminal Cut with the smallest budget. Let E'_{OPT} be this edge set. Since this set disconnects the terminals, it also solves Force Path Cut on $\hat{G}$. In the full proof, we show that there is no way to solve Force Path Cut on $\hat{G}$ with fewer edges. Thus, the solution for Optimal Force Path Cut is the same as the optimal solution to 3-Terminal Cut. In addition, if we consider *any* solution $\hat{E}'$ to Force Path Cut on $\hat{G}$, we show that there is a solution to 3-Terminal Cut that is no larger than $\hat{E}'$. As we discuss in the appendix, this means that the reduction is linear, and thus preserves approximation. Since 3-Terminal Cut is APX-hard, a linear reduction implies that Optimal Force Path Cut is also APX-hard.

To prove that Optimal Force Path Cut is APX-hard for directed graphs as well, we reduce Force Path Cut for undirected graphs to the same problem for directed graphs. Given an undirected graph $G = (V, E)$ and the path p^* from source node s to destination node t, create a new graph $\hat{G} = (V, \hat{E})$ on the same vertex set, with two directed edges for each undirected edge from G; i.e., $\hat{E}$ contains the directed edges (u, v) and (v, u) if and only if E contains the undirected edge $\{u, v\}$. We again consider the case where all weights and all costs are equal. Solve Optimal Force Path Cut on $\hat{G}$, obtaining the solution $\hat{E}'$. For each directed edge in $(u, v) \in \hat{E}'$, remove the undirected edge $\{u, v\}$ from G. In the resulting graph, p^* will be the shortest path from s to t. As we show in Appendix A.2, if the optimal solution includes (u, v), it cannot also include (v, u); i.e., if $(u, v) \in \hat{E}'$, then $(v, u) \notin \hat{E}'$. This implies that the optimal solution size for Force Path Cut in directed graphs is equal to the optimal solution size for Force Path Cut in undirected graphs. By the same argument that we used for undirected graphs, Optimal Force Path Cut is therefore APX-hard for directed graphs.

3.2 Extension to Force Edge Cut and Force Node Cut

The same reduction used to prove that Optimal Force Path Cut is APX-hard can show the same for Optimal Force Edge Cut. The difference is that, rather than letting p^* be the new path from s_1 to s_3, we let e^* be one of the edges along this new path. This will still force the new path from s_1 to s_3 to be the shortest, so the optimal solution will be exactly the same. Like Optimal Force Path Cut, the directed and undirected cases have the same solution, resulting in the following theorem.

Theorem 3.2. *Optimal Force Edge Cut is APX-hard, including the case where all weights and all costs are equal.*

By selecting a node along the new path from s_1 to s_3 to act as v^*, we can make the same argument for Optimal Force Node Cut.

Theorem 3.3. *Optimal Force Node Cut is APX-hard, including the case when all weights and all costs are equal.*

4 PATHATTACK

While Optimal Force Path Cut is APX-hard, its solution can be approximated within a logarithmic factor in polynomial time. This section describes an algorithm to achieve such an approximation.

4.1 Cutting Paths via Set Cover

We first note that Optimal Force Path Cut can be recast as an instance of Weighted Set Cover. In Weighted Set Cover, there is a discrete universe U of elements and a set $\mathcal{S}$ of subsets of U, where each subset $S \in \mathcal{S}$ has an associated cost $c(S) \geq 0$. The objective is to find the subset of $\mathcal{S}$ with the lowest total cost where the union of the elements comprises U, i.e., to find

$$\hat{\mathcal{S}} = \arg\min_{\mathcal{S}' \subseteq \mathcal{S}} \sum_{S \in \mathcal{S}'} c(S) \tag{1}$$

$$\text{s.t.} \bigcup_{S \in \mathcal{S}'} S = U. \tag{2}$$

When we solve Optimal Force Path Cut, the goal is to cut all paths from s to t that are not longer than p^*, and to do so by selecting edges for removal. This is a set cover problem. The universe U is the set of all paths competing to be shortest. This set must be "covered" by removing all such paths from the graph. If any of these paths remains, p^* will not be the shortest path. The subsets in $\mathcal{S}$ are edges: each edge represents the set of paths that use that edge. Cutting the edge is selecting

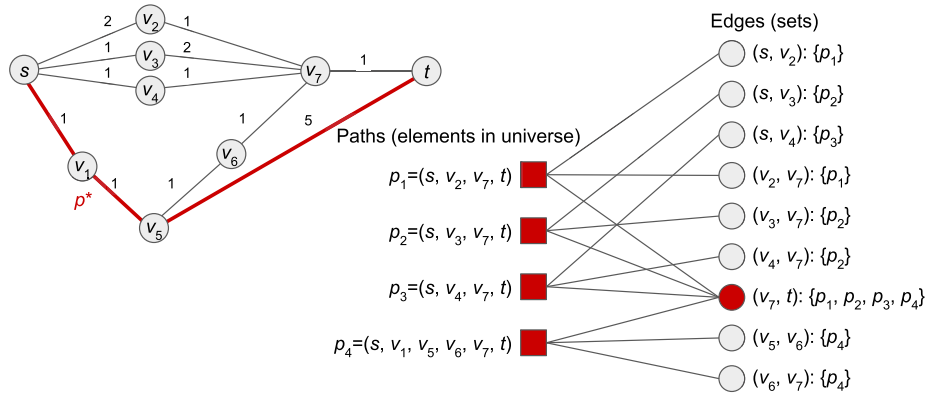

Fig. 2. The Optimal Force Path Cut problem is an example of the Weighted Set Cover problem. In the bipartite graph on the right, the square nodes represent paths and the circle nodes represent edges. Note that edges along p^* are not included. When the red-colored circle (i.e., edge (v_7, t)) is removed, then the red-colored squares (i.e., paths p_1, p_2, p_3, and p_4) are removed.

that set for inclusion in the union, i.e., removing all paths that include the edge. Figure 2 illustrates the connection between the two problems.

Weighted Set Cover not only is APX-hard but also cannot be approximated within less than a logarithmic factor [46]. There are, however, well-known approximation algorithms that enable us to efficiently reach this asymptotic lower bound. The challenge for Force Path Cut is that the universe of competing paths may be extremely large. We address this challenge in the remainder of this section.

4.2 Approximation Algorithms for Fixed Path Sets

We consider two approximation algorithms for Weighted Set Cover, both of which achieve a logarithmic factor approximation for the optimal solution [49]. The first is a greedy method that iteratively selects the most cost-effective set and includes it in the solution; i.e., it selects the set where the number of uncovered elements divided by the cost is the lowest. We likewise iteratively remove the edge that cuts the most uncut paths in P for the lowest cost. We refer to this procedure as `GreedyPathCover`, and we provide its pseudocode in Algorithm 1. We have a fixed set of paths $P \subset P_{p^*}$. Note that this algorithm only uses costs, not weights: the paths of interest have already been determined and we only need to determine the cost of breaking them. `GreedyPathCover` performs a constant amount of work (in expectation, under the simple uniform hashing assumption [15]) at each edge in each path in the initialization loop and the edge and path removal. We use lazy initialization to avoid initializing entries in the tables associated with edges that do not appear in any paths. Thus, populating the tables and removing paths takes time that is linear in the sum of the number of edges over all paths, which we define as

$$M_P := \sum_{p \in P} |E_p|, \tag{3}$$

where E_p is the set of edges along path p. Finding the most cost-effective edge takes $O(M_P)$ time with a naïve implementation, and this portion is run at most once per path, leading to an overall running time as follows.

Proposition 4.1. `GreedyPathCover` *runs in* $O(|P|M_P)$ *time.*

ALGORITHM 1: `GreedyPathCover`: Use a greedy approximation algorithm for Set Cover to identify a set of edges to remove, whose cost is within a logarithmic factor of optimal.

Input: Graph $G = (V, E)$, costs c, target path p^*, path set P, edges E_{keep}
Output: Set E' of edges to cut
$T_P \leftarrow$ empty hash table; `// set of paths for each edge`
$T_E \leftarrow$ empty hash table; `// set of edges for each path`
$N_P \leftarrow$ empty hash table; `// path count for each edge`
foreach $e \in E$ **do**
  $T_P[e] \leftarrow \emptyset$;
  $N_P[e] \leftarrow 0$;
end
foreach $p \in P$ **do**
  $T_E[p] \leftarrow \emptyset$;
  foreach $e \in E_p \setminus E_{\text{keep}}$ **do**
    $T_P[e] \leftarrow T_P[e] \cup \{p\}$;
    $T_E[p] \leftarrow T_E[p] \cup \{e\}$;
    $N_P[e] \leftarrow N_P[e] + 1$;
  end
end
$E' \leftarrow \emptyset$;
while $\max_{e \in E} N_P[e] > 0$ **do**
  $e' \leftarrow \arg\max_{e \in E} N_P[e]/c(e)$; `// find most cost-effective edge`
  $E' \leftarrow E' \cup \{e'\}$;
  foreach $p \in T_P[e']$ **do**
    foreach $e_1 \in T_E[p]$ **do**
      $N_P[e_1] \leftarrow N_P[e_1] - 1$; `// decrement path count`
      $T_P[e_1] \leftarrow T_P[e_1] \setminus \{p\}$; `// remove path`
    end
    $T_E[p] \leftarrow \emptyset$; `// clear edges`
  end
end
return E'

Note that the running time is output sensitive: it is dependent on which paths are selected as constraints over the course of the algorithm. This is the case for all algorithms we propose in this article. Using a more sophisticated data structure, like a Fibonacci heap, to hold the number of paths for each edge would enable finding the most cost-effective edge in constant time, but updating the counts when edges are removed would take $O(\log M_P)$ time, for an overall running time of $O(M_P \log M_P)$. The worst-case approximation factor is the harmonic function of the size of the universe [49], i.e., $H_{|U|} = \sum_{n=1}^{|U|} 1/n$, which implies that the `GreedyPathCover` algorithm has a worst-case approximation factor of $H_{|P|}$.

For the second approximation algorithm, we introduce the integer program formulation of Optimal Force Path Cut, modeled after the formulation for Weighted Set Cover. The objective is to minimize the cost of the edges that are cut. Let $\mathbf{c} \in \mathbb{R}^M_{\geq 0}$ be a vector of edge removal costs, where each entry corresponds to an edge. The binary vector $\mathbf{x}_{\text{cut}} \in \{0, 1\}^M$ indicates the edges to be removed. In the integer program formulation of Weighted Set Cover, each dimension corresponds to a set, and each constraint corresponds to an element. Each element forms a linear constraint, forcing at least one of the sets containing that element to be selected. Let P_{p^*} be the set of paths

that must be cut. For each path $p \in P_{p^*}$, let $\mathbf{x}_p \in \{0,1\}^M$ be the indicator vector for E_p. We solve Optimal Force Path Cut via the integer program

$$\hat{\mathbf{x}}_{\text{cut}} = \arg\min_{\mathbf{x}_{\text{cut}}} \mathbf{c}^\top \mathbf{x}_{\text{cut}} \tag{4}$$

$$\text{s.t. } \mathbf{x}_{\text{cut}} \in \{0,1\}^M \tag{5}$$

$$\mathbf{x}_p^\top \mathbf{x}_{\text{cut}} \geq 1 \ \forall p \in P_{p^*} \tag{6}$$

$$\mathbf{x}_{\text{keep}}^\top \mathbf{x}_{\text{cut}} = 0. \tag{7}$$

Equation (7) prevents any edges in the set E_{keep} from being cut.[2] With target paths, we set $E_{\text{keep}} = E_{p^*}$, but we allow greater flexibility for the methods outlined in Section 5, which target nodes or edges.

The second algorithm is a randomized algorithm that uses this formulation. As with the greedy approach, we focus on a subset of paths $P \subset P_{p^*}$. This algorithm relaxes the integer program (Equations (4)–(7)) by replacing the binary constraint $\mathbf{x}_{\text{cut}} \in \{0,1\}^M$ with a continuous constraint $\mathbf{x}_{\text{cut}} \in [0,1]^M$. When we find the optimal solution $\hat{\mathbf{x}}_{\text{cut}}$ to the relaxed problem, we perform the following randomized rounding procedure for each edge e:

(1) Treat the corresponding entry $\hat{\mathbf{x}}_{\text{cut}}(e)$ as a probability.
(2) Draw $\lceil \ln(4|P|) \rceil$ i.i.d. Bernoulli random variables with probability $\hat{\mathbf{x}}_{\text{cut}}(e)$.
(3) Cut e only if at least one random variable from step 2 is 1.

The resulting graph must be checked against two criteria. First, the constraints must all be satisfied; i.e., all paths in P must be cut. In addition, the cost of cutting the edges must not exceed $\ln(4|P|)$ times the fractional optimum, i.e., the optimal objective of the relaxed **linear program (LP)**. If one of these criteria is not satisfied, the randomized rounding procedure is run again. As shown in detail by Vazirani, each criterion is satisfied with probability at least 3/4 [50]: the probability that any path in P remains uncut is at most 1/4, as is the probability of the solution exceeding $\ln(4|P|)$. Combining these probabilities, the probability of any randomized rounding trial being unsatisfactory is at most 1/2; i.e., the probability of success is greater than or equal to 1/2. This results in the expected number of randomized rounding trials being at most 2. We present pseudocode for this procedure, which we call `RandPathCover`, in Algorithm 2. Its running time is dominated by solving the linear program, which results in the following proposition:

Proposition 4.2. `RandPathCover` *runs in* $\tilde{O}((M_P + |P|^2)|P|^{1/2} + M)$ *time in expectation.*[3]

Proof. A linear program with a sparse constraint matrix $A \in \mathbb{R}^{d \times n}$ (i.e., d constraints, n variables) can be solved in $\tilde{O}((\text{nnz}(A) + d^2)d^{1/2} \log \epsilon^{-1})$ time [33], where $\text{nnz}(\cdot)$ is the number of nonzeros of a sparse matrix. In our case, the number of nonzeros is M_P. The number of constraints is $|P|$, and plugging into the formula, we have $\tilde{O}((M_P + |P|^2)|P|^{1/2})$ (for a fixed level of precision). Initializing $\mathbf{x}_{\text{cut}}$ requires $O(M)$ time. Within the randomized rounding loop, the condition is checked in $O(M)$ time, and the inner loop runs for $O(\log |P|)$ iterations. Within each iteration, edges are selected, and we can consider only those edges where the probability of being selected is nonzero, which leads to $O(M_P)$ trials in each iteration. Creating an indicator vector is also restricted to those edges used in the paths in P, so it takes $O(M_P)$ time as well. Finally, computing not_cut can be performed by a sparse matrix-vector multiplication of the constraint matrix with $\mathbf{x}_{\text{cut}}$, which requires $O(M_P)$ time, and checking for the existence of a violated constraint in the resulting vector,

[2] This could also be achieved by removing these variables from the optimization.
[3] The notation $\tilde{O}$ omits polylogarithmic factors from the O notation.

ALGORITHM 2: `RandPathCover`: Use a relaxed linear program to approximate (within a logarithmic factor) the optimal solution to Set Cover. This identifies a set of edges whose removal disrupts all paths in P.

Input: Graph $G = (V, E)$, costs $\mathbf{c}$, path p^*, path set P, edges E_{keep}
Output: Set E' of edges to cut
$\hat{\mathbf{x}}_{\text{cut}} \leftarrow$ relaxed cut solution to (4)–(7) with paths P;
$\mathbf{x}_{\text{cut}} \leftarrow \mathbf{0}$;
$E' \leftarrow \emptyset$;
not_cut$\leftarrow$ **True**;
while $\mathbf{c}^\top \mathbf{x}_{cut} > \mathbf{c}^\top \hat{\mathbf{x}}_{\text{cut}}(4 \ln(4|P|))$ **or** not_cut **do**
  $E' \leftarrow \emptyset$;
  for $i \leftarrow 1$ *to* $\lceil \ln(4|P|) \rceil$ **do**
    `// randomly select edges based on` $\hat{\mathbf{x}}_{\text{cut}}$
    $E_1 \leftarrow \{e \in E$ with probability $\hat{\mathbf{x}}_{\text{cut}}(e)\}$;
    $E' \leftarrow E' \cup E_1$;
  end
  $\mathbf{x}_{\text{cut}} \leftarrow$ indicator vector for E';
  not_cut$\leftarrow$ ($\exists p \in P$ where $E_p \cap E' \neq \emptyset$);
end
return E'

which requires $O(|P|)$ time. As mentioned earlier, the expected number of randomized rounding trials is at most 2, which only scales the running time of the operations in the loop by a constant. Adding the running times of all components together, we have a total running time of

$$\tilde{O}((M_P + |P|^2)|P|^{1/2}) + O(M + M_P \log |P| + M_P + |P|) = \tilde{O}((M_P + |P|^2)|P|^{1/2} + M). \tag{8}$$

□

4.3 Constraint Generation

Algorithms 1 and 2 work with a fixed set of paths P, but to solve Optimal Force Path Cut, enumeration of the full set of relevant paths may be intractable. Consider, for example, the case where G is a clique (i.e., a complete graph) and all edges have weight 1 except the edge joining s and t, which has weight N. Among all simple paths from s to t, the longest path is the direct path (s, t) that only uses one edge; all other simple paths are shorter, including $(N-2)!$ paths of length $N-1$. Setting $p^* = (s, t)$ makes the full set of constraints (i.e., one for each path that needs to be cut) extremely large. However, if P were to include only those paths of length 2 and 3, we would obtain the optimal solution with only $(N-2)^2 + (N-2)$ constraints. In general, we can solve linear programming problems—even those with infinitely many constraints—as long as we have a polynomial-time method to identify a constraint that a candidate solution violates. In a constraint generation procedure, we iteratively solve a relaxed optimization problem that only considers a subset of the constraints [4]. After solving the relaxed problem, we consult the polynomial-time oracle to determine if any constraint is unsatisfied. If so, this constraint is added and the optimization is run again. The process terminates when the oracle finds that no constraint has been violated.

In our case, the oracle that provides a violated constraint, if one exists, is the shortest path algorithm. After applying a candidate solution, we see if p^* is the shortest path from s to t in the resulting graph. If so, the procedure terminates. If not, the shortest path is added as a new constraint. This procedure works in an iterative process with the approximation algorithms to expand P as new constraints are discovered. We refer to the resulting algorithm as `PATHATTACK` and provide

ALGORITHM 3: PATHATTACK: Use constraint generation combined with the approximation algorithms GreedyPathCover or RandPathCover to approximately solve Optimal Force Path Cut.

Input: Graph $G = (V, E)$, costs c, weights w, target path p^*, edges E_{keep}, cover alg. Cvr
Output: Set E' of edges to cut
$E' \leftarrow \emptyset$;
$P \leftarrow \emptyset$;
$\mathbf{c} \leftarrow$ vector from costs $c(e)$ for $e \in E$;
$G' \leftarrow (V, E \setminus E')$;
$s, t \leftarrow$ source and destination nodes of p^*;
$p \leftarrow$ shortest path from s to t in G' (not including p^*);
while *p is not longer than p^** **do**
  $P \leftarrow P \cup \{p\}$;
  $E' \leftarrow$Cvr$(G, \mathbf{c}, p^*, P)$;
  $G' \leftarrow (V, E \setminus E')$;
  $p \leftarrow$ shortest path from s to t in G' (not including p^*) using weights w;
end
return E'

pseudocode in Algorithm 3. Depending on whether we use GreedyPathCover or RandPathCover, we refer to the algorithm as PATHATTACK-Greedy or PATHATTACK-Rand, respectively.

The running time of PATHATTACK is computed as follows. The initialization takes $O(M)$ time. Within the constraint generation loop, a path is added to P (time proportional to its length), the Set Cover approximation algorithm is run (given by Propositions 4.1 and 4.2), edges are removed (at most M_P), and a new shortest path is computed. We may have to compute the second shortest path, but we use Eppstein's algorithm for k shortest paths, even though it allows loopy paths: if the second shortest path has a cycle, then p^* is the unique shortest path. This runs in $O(M + N \log N)$ time [22]. Checking path lengths costs at most $O(N)$ time. The constraint generation loop is run $|P|$ times. Combining all terms, we have the following asymptotic running times.

Proposition 4.3. PATHATTACK-Greedy *and* PATHATTACK-Rand *run in* $O(|P|^2 M_P)$ *time and* $\tilde{O}((M_P + |P|^2)|P|^{3/2} + |P|M)$ *time, respectively.*

4.4 PATHATTACK Convergence and Approximation Guarantees

While the approximation factor for Set Cover is a function of the size of the universe (all paths that need to be cut), this is not the fundamental factor in the approximation in our case. The approximation factor for PATHATTACK-Greedy is based only on the paths we consider explicitly. Using only a subset of constraints, the worst-case approximation factor is determined by the size of that subset. By the final iteration of PATHATTACK, however, we have a solution to Force Path Cut with a budget within a factor of $H_{|P|}$ of the minimum, using $|P|$ from the final iteration. This yields the following proposition:

Proposition 4.4. *The approximation factor of* PATHATTACK-Greedy *is at most* $H_{|P|}$ *times the solution to Optimal Force Path Cut.*

The worst-case approximation factor for PATHATTACK-Rand is logarithmic by construction:

Proposition 4.5. PATHATTACK-Rand *yields a worst-case* $O(\log |P|)$ *approximation to Optimal Force Path Cut.*

There is also a variation of PATHATTACK-Rand that can guarantee polynomial-time convergence. While the number of implicit constraints may be extremely large, each one is a standard linear

inequality constraint, which implies that the feasible region is convex. Given a polynomial-time oracle that returns a constraint violated by a proposed solution $\hat{\mathbf{x}}_{\text{cut}} \in [0,1]^M$ to the relaxed LP, we could use Khachiyan's ellipsoid algorithm (see [24]), which can solve a linear program with an arbitrary (or infinite) number of linear constraints in a polynomial number of iterations [25]. We use the randomized rounding procedure from Algorithm 2 to achieve such an oracle. In Appendix B, we prove that this oracle terminates in polynomial time with high probability. This results in the following theorem.

Theorem 4.6. PATHATTACK-Rand *using the ellipsoid algorithm converges in polynomial time with probability at least* $1 - (1/|E|)$.

While this demonstrates that it is not NP-hard to solve Optimal Force Path Cut within a logarithmic approximation and shows that there is a version of PATHATTACK that is guaranteed to run in polynomial time, we do not use the ellipsoid algorithm in our experiments, as other methods converge much faster in practice. As discussed in Section 6, we use the Gurobi optimization package to solve all optimization problems, and this software uses a combination of simplex and barrier methods for continuous linear programs. These methods do not, however, guarantee polynomial-time convergence when using constraint generation with a set of constraints that is nonpolynomial in size.

4.5 PATHATTACK for Node Removal

This article is focused on the case where edges are removed from a graph, but there are applications in which removing nodes is more appropriate. In a computer network, for example, it may be more likely for a node to be taken offline (e.g., via a denial-of-service attack) rather than to have particular connections disallowed. Force Path Cut via node removal is a more complicated problem; for a given p^*, there is not always a solution, regardless of the budget constraint. Consider, for example, a triangle of nodes u, v, and w, where all weights are equal, and $p^* = (u, v, w)$: none of the three nodes can be removed because it would destroy p^*, so p^* cannot be made the shortest path due to the existence of the edge connecting u to w.

In cases where it is possible, however, we can still use PATHATTACK. The mapping to Weighted Set Cover is analogous: a path is an element and a node corresponds to the set of paths from s to t using that node. Given a graph and p^*, we can check if it is possible to make p^* the shortest path from s to t via node removal: if p^* is the shortest path from s to t in the induced subgraph of the nodes on p^*, then there is a solution. If not, there is none; the method to check has demonstrated that, even if all other nodes are removed, p^* is not the shortest path. If there is a possible solution, we can run PATHATTACK for node removal and achieve the same convergence and approximation guarantees as we have for edge removal. As this is ancillary to the main focus of the article, we relegate further discussion of node removal to Appendix C.

4.6 Limited Attacker Capability

In practice, it is likely that there will be some edges the attacker will not be able to remove. We can simply integrate this into the optimization formulation by setting the removal costs of these edges to infinity. In the linear program formulation, this can be accomplished by omitting the columns associated with the infinite-cost edges in Equations (4) through (7). As with the node removal case, this will require a check to ensure a solution exists, which can be achieved by removing all edges available to the attacker—other than those that are part of p^*—and observing whether p^* becomes the shortest path between its endpoints. If so, PATHATTACK can be used within the restricted edge set to find an approximately optimal solution.

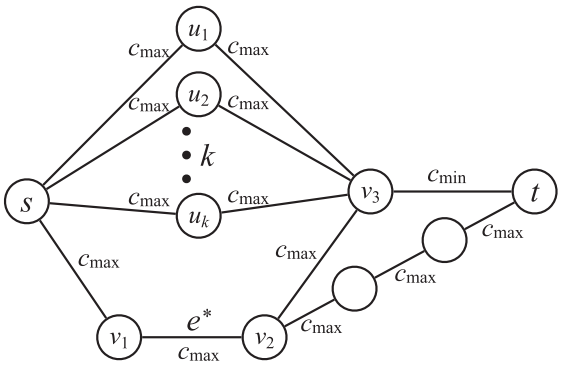

Fig. 3. A scenario where preserving the initial shortest path through e^* results in a bad solution. Here, all edges have the same weight and the labels are removal costs. The shortest path through e^* is (s, v_1, v_2, v_3, t) and includes the edge $\{v_3, t\}$ with cost $c_{\min}$. However, if we solve Optimal Force Path Cut with this path as p^*, the k parallel paths on the top of the figure need to be cut, resulting in a cost of $kc_{\max}$. If, on the other hand, $\{v_3, t\}$ were removed, then the shortest path from s to t would go through e^*, at a factor of $k\frac{c_{\max}}{c_{\min}}$ lower cost. If k is $\Omega(M)$, the optimal solution preserving (s, v_1, v_2, v_3, t) is within a constant factor of removing all edges, when a solution with the lowest possible cost was available.

5 HEURISTICS FOR FORCE EDGE CUT AND FORCE NODE CUT

Solving Optimal Force Edge Cut or Optimal Force Node Cut requires an additional layer of optimization. Since these problems do not consider a fixed path, there is the possibility that a candidate solution could be improved by cutting the current shortest path through the target. Take, for example, the graph in Figure 3. The shortest path from s to t via e^*—i.e., (s, v_1, v_2, v_3, t)—is four hops and uses the sole low-cost edge. If we solve Optimal Force Path Cut with this path as p^*, all of the k paths along the top of the figure have to be cut. If we remove the low-cost edge, all these paths would be cut and the shortest path from s to t will go through e^*. Since that edge is being preserved as part of p^*, however, each of the k parallel two-hop paths has to be cut individually and incur a cost of $c_{\max}$: if any of these paths remains, it is shorter than (s, v_1, v_2, v_3, t), so e^* is not on the shortest path. For large k, we see that this cost is within a constant factor of removing all edges in the graph, when a low-cost solution was available. More concretely, note that $M = 2k + 7$. The cost of the solution that preserves p^* is

$$kc_{\max} = \frac{M-7}{2}c_{\max} = \Omega(Mc_{\max}). \tag{9}$$

Without the requirement to preserve p^*, we could remove $\{v_3, t\}$ and incur a cost of $c_{\min}$, which is the minimum possible cost for an attack that removes more than zero edges. The existence of this scenario proves the following theorem.

Theorem 5.1. *Solving Optimal Force Path Cut targeting the shortest path through e^* (or v^*) may yield a cost $\Omega(M\frac{c_{max}}{c_{min}})$ times greater than solving Optimal Force Edge Cut (or Optimal Force Node Cut), where c_{max} and c_{min} are, respectively, the maximum and minimum edge removal costs.*

To both minimize cost and allow the flexibility to alter the target path, we formulate a nonconvex optimization problem. We start with a formulation of a linear program to obtain the shortest path through e^* (or v^*). Recall the linear program formulation of the shortest path problem (see, e.g., [3]). Using a graph's incidence matrix, finding the shortest path can be formulated as a linear program. In the incidence matrix, denoted by $\mathbf{C}$, each row corresponds to a node and each column corresponds to an edge. The column representing the edge from node u to node v contains -1

in the row for u, 1 in the row for v, and zeros elsewhere. If the graph is undirected, the edge orientation is arbitrary. To identify the shortest path from s to t, define the vector $\mathbf{d} \in \{-1, 0, 1\}^N$, which is -1 in the row corresponding to s, 1 in the row for t, and zeros elsewhere. The shortest path is the solution to the integer linear program

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \mathbf{x}^\top \mathbf{w} \tag{10}$$

$$\text{s.t. } \mathbf{x} \in \{0, 1\}^M \tag{11}$$

$$\mathbf{C}\mathbf{x} = \mathbf{d}. \tag{12}$$

The resulting vector $\hat{\mathbf{x}}$ is an indicator for the edges in the shortest path.[4] Equation (12) ensures that $\hat{\mathbf{x}}$ is a path, and the objective guarantees that the result has minimum weight. Due to the structure of the incidence matrix, we can relax Equation (11) into the continuous interval $\mathbf{x} \in [0, 1]^M$. If the shortest path from s to t is unique, $\hat{\mathbf{x}}$ will be an indicator vector for the shortest path despite this relaxation. If there are multiple shortest paths, $\hat{\mathbf{x}}$ will be a convex combination of the vectors for these paths.

Recall that the goal of the attacker is to force travelers from s to t to traverse the edge e^*, so we alter the formulation to find such a path. To obtain the shortest path through e^*, we perform a similar optimization over two paths: the path from s to e^* and the path from e^* to t.[5] Let $e^* = (u, v)$ be the target edge. (In the undirected setting, we consider both (u, v) and (v, u) independently and choose the lower-cost solution.) Since we consider two paths, we have two vectors $\mathbf{d}_1$ and $\mathbf{d}_2$, analogous to $\mathbf{d}$ in Equation (12). Assuming that $s \neq u$ and $t \neq v$, $\mathbf{d}_1$ is -1 in the row of s and 1 in the row of u, with zeros elsewhere, and $\mathbf{d}_2$ similarly has -1 in the row for v and 1 in the row for t. (If $s = u$, $\mathbf{d}_1$ is all zeros, as is $\mathbf{d}_2$ if $t = v$.) Since we are optimizing two paths, we use two path vectors, $\mathbf{x}_1 \in [0, 1]^M$ and $\mathbf{x}_2 \in [0, 1]^M$.

It is not, however, sufficient to obtain a path from s to u and one from v to t: the concatenation must also be a simple (acyclic) path. (Otherwise a shortest path algorithm would excise the cycle.) To ensure the resulting path has no cycles, we add a constraint to ensure any node is visited at most once. Since the path vectors are optimized over edges, we ensure any node occurs at most twice in the set of edges, aside from the terminal nodes, which occur at most once. To obtain this constraint, we use the matrix $\mathbf{C}_{\text{abs}}$, which contains the absolute values of the incidence matrix; i.e., the ith row and jth column of $\mathbf{C}_{\text{abs}}$ contains $|\mathbf{C}_{ij}|$. We also define $\mathbf{d}_{\text{abs}} \in \{1, 2\}^N$ to be 1 in the rows associated with s, t, u, and v, and 2 elsewhere. This yields the linear program

$$\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2 = \arg\min_{\mathbf{x}_1, \mathbf{x}_2} (\mathbf{x}_1 + \mathbf{x}_2)^\top \mathbf{w} \tag{13}$$

$$\text{s.t. } \mathbf{x}_1, \mathbf{x}_2 \in [0, 1]^M \tag{14}$$

$$\mathbf{C}\mathbf{x}_1 = \mathbf{d}_1 \tag{15}$$

$$\mathbf{C}\mathbf{x}_2 = \mathbf{d}_2 \tag{16}$$

$$\mathbf{C}_{\text{abs}}(\mathbf{x}_1 + \mathbf{x}_2) \leq \mathbf{d}_{\text{abs}}. \tag{17}$$

To solve Force Edge Cut, we use the same constraint generation technique as in PATHATTACK, plus a nonlinear constraint to ensure the returned path does not contain any removed edges. As in PATHATTACK, we use a vector $\mathbf{x}_{\text{cut}} \in [0, 1]$. In addition to constraining $\mathbf{x}_{\text{cut}}$ to not cut e^*, we add

[4] In the undirected case, to denote traversal of an edge in the opposite direction of its arbitrary orientation, we consider $\mathbf{x}_{\text{pos}}$ and $\mathbf{x}_{\text{neg}}$, with Equation (12) replaced with $\mathbf{C}(\mathbf{x}_{\text{pos}} - \mathbf{x}_{\text{neg}}) = \mathbf{d}$ and the objective replaced with $(\mathbf{x}_{\text{pos}} + \mathbf{x}_{\text{neg}})^\top \mathbf{w}$. To restrict traversing an edge in both directions, we add the constraint $\mathbf{x}_{\text{pos}} + \mathbf{x}_{\text{neg}} \leq \mathbf{1}$

[5] For brevity, we focus on the solution to Optimal Force Edge Cut. The formulation for Optimal Force Node Cut is similar.

the nonconvex bilinear constraint that $\mathbf{x}_{\text{cut}}$ is 0 anywhere $\mathbf{x}_1$ or $\mathbf{x}_2$ is nonzero. Finally, we again consider a subset of paths P we want to ensure are cut. The resulting nonconvex program

$$\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \hat{\mathbf{x}}_{\text{cut}} = \underset{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_{\text{cut}}}{\arg\min} (\mathbf{x}_1 + \mathbf{x}_2)^\top \mathbf{w} \tag{18}$$

$$\text{s.t. } \mathbf{x}_1, \mathbf{x}_2 \in [0,1]^M \tag{19}$$

$$\mathbf{x}_{\text{cut}} \in \{0,1\}^M \tag{20}$$

$$\mathbf{C}\mathbf{x}_i = \mathbf{d}_i,\ i \in \{1,2\} \tag{21}$$

$$\mathbf{C}_{\text{abs}}(\mathbf{x}_1 + \mathbf{x}_2) \leq \mathbf{d}_{\text{abs}} \tag{22}$$

$$\mathbf{x}_i^\top \mathbf{x}_{\text{cut}} = 0,\ i \in \{1,2\} \tag{23}$$

$$\mathbf{x}_{\text{cut}}(e^*) = 0 \tag{24}$$

$$\mathbf{x}_{\text{cut}}^\top \mathbf{c} \leq b \tag{25}$$

$$\mathbf{x}_{\text{cut}}^\top \mathbf{x}_p \geq 1\ \forall p \in P \tag{26}$$

provides a partial solution (i.e., a solution for a subset of competing paths).

The constraint generation mechanism differs slightly from the Force Path Cut case. When solving Force Path Cut, there is a specific target path whose length does not change. For Force Edge Cut, when a new path is added to the constraint set P, it may change the shortest uncut path through e^*, and thus change the length threshold for inclusion. Thus, we want P to include all paths that are not longer than the shortest path in the solution, which is not available until the problem has been solved. As in PATHATTACK, each time we solve the optimization problem, we find the shortest path after removing the edges indicated by $\hat{\mathbf{x}}_{\text{cut}}$ as well as e^*. If this path is not longer than the shortest uncut path through e^*—the path indicated by $\hat{\mathbf{x}}_1$, $\hat{\mathbf{x}}_2$, and e^*—then this alternative path must also be cut, and we add a constraint to achieve this. Algorithm 4 provides psuedocode for this modified constraint generation procedure.

We note that solving for Equation (18) is a nonconvex mixed-integer optimization that is solved using branch and cut, which has exponential running time in the worst case. We therefore set a maximum running time $t_{\max}$ after which we choose the best solution found so far. We express the algorithm running times in terms of this quantity, with the understanding that it stands in for the truncated running time of an algorithm that has worst-case exponential running time. The constraint generation procedure has a running time as follows.

Proposition 5.2. *Algorithm 4 runs in* $O(|P|(t_{\max} + M + N \log N))$ *time.*

Proof. Instantiating $\mathbf{C}$, $\mathbf{w}$, $\mathbf{c}$, $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{C}_{\text{abs}}$, and $\mathbf{d}_{\text{abs}}$ costs $O(M + N)$ time. Within the loop, we first get the objective from Equation (18), which runs for at most $t_{\max}$ time. Extracting a single path from $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$ takes $O(M)$ (any edge is considered at most twice). Extracting p_1 from the indicator vectors takes $O(|E_{p_1}|)$ time. The size of E' is $O(M)$, and creating G' takes $O(N + M)$ time. Finding the shortest path in G' takes $O(M + N \log N)$ time, and adding a new constraint to P takes $O(|E_{p_2}|)$ time. Combining all operations in the loop, we have

$$O(t_{\max} + M + N + N \log N + |E_{p_1}| + |E_{p_2}|) = O(t_{\max} + M + N \log N). \tag{27}$$

The number of iterations of the loop is $|P|$, and the asymptotic running time within the loop is greater than the time required to instantiate the variables before the loop. This yields an overall running time of $O(|P|(t_{\max} + M + N \log N))$. □

To minimize the cost of the attack, we perform a binary search with respect to the budget b. We obtain upper and lower bounds for the budget by running PATHATTACK targeting the shortest path

ALGORITHM 4: Constraint generation procedure for Force Edge Cut: iteratively solve a nonconvex optimization problem for the current set of constraints, and add a new constraint if the current solution is not feasible.

Input: Graph $G = (V, E)$, weights w, costs c, source s, destination t, target edge e^*, budget b
Output: Set E' of edges to cut
$\mathbf{C} \leftarrow$ unweighted incidence matrix of G;
$\mathbf{w} \leftarrow$ weight vector from w; $\mathbf{c} \leftarrow$ cost vector from c;
$\mathbf{d}_1 \leftarrow s$ to e^* vector; $\mathbf{d}_2 \leftarrow e^*$ to t vector;
$\mathbf{C}_{\text{abs}} \leftarrow |\mathbf{C}|$; $\mathbf{d}_{\text{abs}} \leftarrow$ "no cycle" vector;
$P \leftarrow \emptyset$;
done $\leftarrow$ **False**;
while not done **do**
  $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \hat{\mathbf{x}}_{\text{cut}} \leftarrow$ solution to (18);
  $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2 \leftarrow$ extract single path indicator from $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2$ if not binary;
  $p_1 \leftarrow$ path from $\hat{\mathbf{x}}_1$, e^*, and $\hat{\mathbf{x}}_2$;
  $E' \leftarrow$ edges with nonzeros in $\hat{\mathbf{x}}_{\text{cut}}$;
  $G' \leftarrow (V, E \setminus (E' \cup \{e^*\}))$;
  $p_2 \leftarrow$ shortest path from s to t in G' (using weights w);
  if *p_2 is not longer than p_1* **then**
    $P \leftarrow P \cup \{p_2\}$;
  else
    done $\leftarrow$ **True**;
  end
end
return E';

through e^*. We run the standard PATHATTACK to get the upper bound, and remove the constraint that the target path is uncut for the lower bound, instead only constraining that e^* is not cut. If, during the search, a new upper bound is discovered (i.e., a path through e^* is discovered that requires a budget smaller than the one under consideration), we create new upper and lower bounds based on the new satisfactory path. Algorithm 5 outlines this procedure.

The binary search procedure in Algorithm 5 will terminate in a number of iterations that is logarithmic in the cost $c_{\text{total}} = \sum_{e \in E} c(e)$. Within the loop, we run Algorithm 4, which by Proposition 5.2 runs in $O(|P|(t_{\max} + M + N \log N))$ time. In the case that Algorithm 4 cannot be solved, we remove constraints, which takes $O(M_P) = O(|P|N)$ time. Otherwise, we create a new graph ($O(N + M)$ time), find the shortest path ($O(M + N \log N)$ time), run PATHATTACK twice ($\tilde{O}((M_P + |P|^2)|P|^{3/2} + |P|M)$ time, by Proposition 4.3), and compute new budgets ($O(M)$ time). Combining all terms within the loop, we have a running time of

$$\begin{aligned} &O(|P|(t_{\max} + M + N \log N) + M + M_P + N + N \log N) + \tilde{O}((M_P + |P|^2)|P|^{3/2} + |P|M) \\ &\qquad = \tilde{O}((M_P + |P|^2)|P|^{3/2} + |P|(t_{\max} + M + N)), \end{aligned} \tag{28}$$

with logarithmic factors being dropped due to the $\tilde{O}$ notation. The number of iterations in the loop adds a logarithmic factor to the running time, which is also omitted within the notation, yielding the following proposition.

Proposition 5.3. *Algorithm 5 runs in $\tilde{O}((M_P + |P|^2)|P|^{3/2} + |P|(t_{\max} + M + N))$ time.*

In addition to the combinatorial optimization method, we consider a heuristic algorithm that seeks to identify the bottlenecks that prevent an optimal solution, as illustrated in Figure 3, and

ALGORITHM 5: Combinatorial search for Optimal Force Edge Cut: perform a binary search over the attack budget, applying Algorithm 4 at each candidate budget.

Input: Graph $G = (V, E)$, weights w, costs c, source s, destination t, target edge e^*, tol. ϵ
Output: Set E' of edges to cut
$p \leftarrow$ shortest path from s to t via e^*;
$E_p \leftarrow$ edges in p;
$E_{\text{upper}} \leftarrow$PATHATTACK$(G, c, w, p, E_p,$ RandPathCover$)$;
$b_{\text{upper}} \leftarrow \sum_{e \in E_{\text{upper}}} c(e)$;
$E_{\text{lower}} \leftarrow$PATHATTACK$(G, c, w, p, \{e^*\},$ RandPathCover$)$;
$b_{\text{lower}} \leftarrow \sum_{e \in E_{\text{lower}}} c(e)$;
while $b_{\text{upper}} - b_{\text{lower}} > \epsilon$ **do**
  $b_{\text{mid}} \leftarrow (b_{\text{upper}} + b_{\text{lower}})/2$;
  $E_{\text{mid}} \leftarrow$Algorithm 4$(G, w, c, s, t, b_{\text{mid}})$;
  if *Algorithm 4 could not be solved* **then**
    $b_{\text{lower}} \leftarrow b_{\text{mid}}$; // budget is too small
    remove constraints accrued since last iteration
  else
    // budget is sufficient
    $G' \leftarrow (V, E \setminus E_{\text{mid}})$;
    $p \leftarrow$ shortest path from s to t in G'; // this path will include e^*
    $E_{\text{temp}} \leftarrow$PATHATTACK$(G, c, w, p, E_p,$ RandPathCover$)$;
    $b_{\text{temp}} \leftarrow \sum_{e \in E_{\text{temp}}} c(e)$;
    if $b_{\text{temp}} < b_{\text{mid}}$ **then**
      $b_{\text{upper}} \leftarrow b_{\text{temp}}$;
      $E_{\text{upper}} \leftarrow E_{\text{temp}}$;
    else
      $b_{\text{upper}} \leftarrow b_{\text{mid}}$;
      $E_{\text{upper}} \leftarrow E_{\text{mid}}$;
    end
    $E_{\text{temp}} \leftarrow$PATHATTACK$(G, c, w, p, \{e^*\},$ RandPathCover$)$;
    $b_{\text{temp}} \leftarrow \sum_{e \in E_{\text{temp}}} c(e)$;
    if $b_{\text{temp}} > b_{\text{lower}}$ **then**
      $b_{\text{lower}} \leftarrow b_{\text{temp}}$;
      $E_{\text{lower}} \leftarrow E_{\text{temp}}$;
    end
  end
end
return E_{upper}; // return the best valid solution found

that is guaranteed to run in polynomial time. As in the combinatorial optimization, we leverage PATHATTACK with and without constraints to avoid cutting the target path. In this case, after running PATHATTACK while only preventing e^* from being cut, we consider the edges on the target path that are cut by PATHATTACK. For each of these edges, we consider the possibility of either (1) removing the edge from the graph entirely or (2) marking it to never be removed. If removal of any of these edges causes t to become unreachable from s, we add that edge to a list of uncuttable edges. Otherwise, we consider the case in which each of these edges is removed, each time finding the shortest path through e^*. We run PATHATTACK in each case and find the edge whose removal results in the lowest upper bound on the removal budget. This edge is

ALGORITHM 6: Heuristic search for Optimal Force Edge Cut: iteratively solve PATHATTACK and identify bottleneck edges to ignore.

Input: Graph $G = (V, E)$, weights w, costs c, source s, destination t, target edge e^*, tol. ϵ
Output: Set E' of edges to cut
$E_{\text{always}} \leftarrow \emptyset$;
$E_{\text{never}} \leftarrow \emptyset$;
$E_{\text{best}} \leftarrow \emptyset$;
$c_{\text{best}} \leftarrow \infty$;
repeat
  $G' \leftarrow (V, E \setminus E_{\text{always}})$;
  $p \leftarrow$ shortest path from s to t via e^* in G';
  $E_p \leftarrow$ edges in p;
  $E_{\text{upper}} \leftarrow$PATHATTACK$(G, c, w, p, E_p,$ RandPathCover$)$;
  $b_{\text{upper}} \leftarrow \sum_{e \in E_{\text{upper}}} c(e)$;
  $E_{\text{lower}} \leftarrow$PATHATTACK$(G', c, w, p, \{e^*\} \cup E_{\text{never}},$ RandPathCover$)$;
  $b_{\text{lower}} \leftarrow \sum_{e \in E_{\text{lower}} \cup E_{\text{always}}} c(e)$;
  if $b_{\text{upper}} < c_{\text{best}}$ **then**
    $c_{\text{best}} \leftarrow b_{\text{upper}}$;
    $E_{\text{best}} \leftarrow E_{\text{upper}}$;
  end
  if $b_{\text{lower}} < b_{\text{upper}}$ **then**
    budget $\leftarrow$ empty hash table;
    for $e \in E_p \cap E_{\text{lower}}$ **do**
      remove e from G;
      $E_1 \leftarrow$PATHATTACK$(G, c, w, p, E_p,$ RandPathCover$)$;
      budget$[e] \leftarrow c(e) + \sum_{e_1 \in E_1} c(e_1)$;
      add e to G;
    end
    if $\exists e \in E_p \cap E_{\text{lower}}$ *where removing* e *disconnects* s *and* t **then**
      $E_{\text{never}} \leftarrow E_{\text{never}} \cup \{e\}$;
    else
      $e_{\text{new}} \leftarrow \arg\min_{e \in E_p \cap E_{\text{lower}}}$ budget$[e]$;
      $E_{\text{always}} \leftarrow E_{\text{always}} \cup \{e\}$;
    end
  end
until $c_{\text{best}} \leq b_{\text{lower}} + \epsilon$;
return E_{best};

added to a list of edges that will always be removed. This procedure terminates when the upper and lower bounds converge. A pseudocode description of this heuristic search is provided in Algorithm 6.

In each iteration of Algorithm 6, PATHATTACK is run as many as N times (once for every edge in p, the shortest path from s to t through e^*), yielding a total running time of $\tilde{O}(N((M_P + |P|^2)|P|^{3/2} + |P|M))$. This dominates the interior of the loop, with the exception of finding p: this also involves solving a sparse linear program with $O(M)$ nonzeros and $O(N)$ constraints. Following the same formula as we used for PATHATTACK, this results in a running time of $\tilde{O}((M + N^2)N^{1/2})$. The overall number of iterations in the loop is bounded by M (an edge is removed every iteration that a solution is not found), which results in an overall running time as follows.

Proposition 5.4. *Algorithm 6 runs in* $\tilde{O}(MN((M_P + |P|^2)|P|^{3/2} + |P|M) + (M + N^2)MN^{1/2})$ *time.*

As with PATHATTACK, this guarantees a polynomial running time if the ellipsoid algorithm is used, but in our experiments we use other methods that are faster in practice.

6 EXPERIMENTS

This section presents baselines, datasets, experimental setup, and results.

6.1 Baseline Methods

We consider two simple greedy methods as baselines for assessing performance of PATHATTACK. Each of these algorithms iteratively computes the shortest path p between s and t; if p is not longer than p^*, it uses some criterion to cut an edge from p. When we cut the edge with minimum cost, we refer to the algorithm as GreedyCost. We also consider a version based on the most vital edge of the current shortest path [44], which is the edge that, if removed, results in the greatest distance between the endpoints. If p is the current shortest path from s to t, the score of edge e is the distance from s to t if e were removed divided by the cost of e. We iteratively remove the edge with the highest score until p^* is the shortest path. This version of the algorithm is called GreedyMVE. In either case, edges from p^* are not allowed to be cut. (In prior work [42], we also considered using the eigenscore of each edge [48], but this consistently underperformed GreedyCost.) We provide pseudocode that encompasses both baselines in Algorithm 7. The baseline method to solve Optimal Force Edge Cut (or Optimal Force Node Cut) is to identify the shortest path through e^* (or v^*), use this path as p^*, and solve PATHATTACK.

In each iteration of outer loop in Algorithm 7, we find up to two shortest paths, with a running time of $O(M + N \log N)$. For each edge in the shortest path, we either do a constant amount of work (in the case of GreedyCost) or find a new distance from s to t (for GreedyMVE), which requires $O(M + N \log N)$. In the case of GreedyMVE, this occurs once for each edge on all paths we identify as constraints, defined as M_P in Equation (3). This results in the following asymptotic running times.

Proposition 6.1. GreedyCost *runs in* $O(|P|(M + N \log N))$ *time and* GreedyMVE *runs in* $O(M_P(M + N \log N))$ *time.*

6.2 Synthetic and Real Networks

We use both synthetic and real networks in our experiments. For the synthetic networks, we run seven different graph models to generate 100 synthetic networks of each model. We pick parameters to yield networks with similar numbers of edges ($\approx$ 160K). We use 16,000-node Erdős–Rényi graphs, both undirected (ER) and directed (DER), with edge probability 0.00125; 16,000-node **Barabási–Albert (BA)** graphs with average degree 20; 16,000-node **Watts–Strogatz (WS)** graphs with average degree 20 and rewiring probability 0.02; 2^{14}-node stochastic **Kronecker (KR)** graphs; 285×285 **lattices (LAT)**; and 565-node **complete (COMP)** graphs.

We use seven weighted and unweighted networks. The unweighted networks are the **Wikispeedia (WIKI)** graph [53], an Oregon **autonomous system (AS)** network [34], and a **Pennsylvania road network (PA-ROAD)** [35]. The weighted networks are **Central Chilean Power Grid (GRID)** [32], **Lawrence Berkeley National Laboratory network data (LBL)**, the **Northeast US Road Network (NEUS)**, and the **DBLP coauthorship graph (DBLP)** [5]. All real networks are undirected except for WIKI and LBL. The networks range from 444 edges on 347 nodes to over 8.3M edges on over 1.8M nodes, with average degree ranging from 2.5 to 46.5 and triangle count ranging from 40 to nearly 27M. Further details on the real and synthetic networks—including URLs to the real data—are provided in Appendix D.

ALGORITHM 7: Baseline algorithm: implement `GreedyCost` or `GreedyMVE` depending on the parameter "edge_metric." Both methods iteratively choose an edge to remove from the current shortest path until the target path p^* is the shortest.

Input: Graph $G = (V, E)$, weights w, costs c, target path p^*, string edge_metric
Output: Set E' of edges to cut
$s \leftarrow$ first node of p^*;
$t \leftarrow$ last node of p^*;
$E' \leftarrow \emptyset$;
$p \leftarrow$ shortest path from s to t in $G' = (V, E \setminus E')$ with weights w (other than p^*);
while p^* *is not shorter than* p **do**
  best_score $\leftarrow 0$;
  $e_1 \leftarrow \emptyset$;
  for $e \in E_p \setminus E_{p^*}$ **do**
    current_score $\leftarrow 1$;
    if *edge_metric* = `''MVE''` **then**
      current_score $\leftarrow$ distance from s to t in $G'' = (V, E \setminus (E' \cup \{e\}))$ with weights w;
    end
    current_score$\leftarrow$current_score$/c(e)$;
    if *current_score* > *best_score* **then**
      best_score $\leftarrow$ current_score;
      $e_1 \leftarrow e$;
    end
  end
  $E' \leftarrow E' \cup \{e_1\}$;
  $p \leftarrow$ shortest path from s to t in $G' = (V, E \setminus E')$ with weights w (other than p^*);
end
return E';

For the synthetic networks and unweighted real networks, we consider three different edge-weight assignment schemes with different levels of entropy: uniform random weights (high entropy), Poisson random weights (lower entropy), or equal weights (no entropy). For Poisson weights, each edge e has an independently random weight $w_e = 1 + w'_e$, where w'_e is drawn from a Poisson distribution with rate parameter 20. For uniform weights, each weight is drawn from a discrete uniform distribution of integers from 1 to 41. This yields the same average weight as Poisson weights.

6.3 Experimental Setup

For each graph—considering graphs with different edge-weighting schemes as distinct—we run 100 experiments. In each experiment, we select s and t uniformly at random among all nodes, with the exception of LAT, PA-ROAD, and NEUS, where we select s uniformly at random and select t at random among nodes 30 to 50 hops away from s, where the distance is selected uniformly at random from within the range.[6] Given s and t, we identify the shortest simple paths and use the 100th, 200th, 400th, and 800th shortest as p^* in four experiments. For the large grid-like networks (LAT, PA-ROAD, and NEUS), this procedure is run using only the 60-hop neighborhood of s. We focus on the case where the edge removal cost is equal to the weight (distance). For Force Edge Cut

[6]This alternative method of selecting the destination was used due to the computational expense of identifying successive shortest paths in large grid-like networks.

and Force Node Cut, we consider consecutive shortest paths from s to t until we see five edges (or nodes) not on the initial shortest path. The fifth edge (or node) we see that was not on the original shortest path is used as e^* (or v^*).

When running Algorithm 4, we let the nonconvex optimization (18) run for no more than 10 minutes. If a feasible point is not found in that time, we assume the model is infeasible and increase the budget. We stop the procedure every 30 seconds to check the best candidate solution and see if it satisfies our criteria. In addition, we stop the optimization if the objective matches the length of our best incumbent solution. If we consider a budget for over 8 hours, we consider it infeasible and increase the lower bound. The entire procedure is terminated, and the lowest upper bound returned, if it is still running after 24 hours.

The experiments were run on Linux machines with 32 cores and 192 GB of memory. The LP in PATHATTACK-Rand was implemented using Gurobi 9.1.1, and shortest paths were computed using `shortest_simple_paths` in NetworkX.[7] The combinatorial search method is set to use four threads.

6.4 Pathattack Results

We treat the result of GreedyCost as our baseline cost and report the cost of other algorithms' solutions as a reduction from that baseline. In all experiments, we set edge removal costs equal to the weights. Figure 4 shows the results on both synthetic and real unweighted graphs, which have had synthetic weights added to the edges. Figure 5 shows the results on real weighted networks. In these figures, the 800th shortest path is used as p^*; other results were similar and omitted for brevity.

Considering the difference between the two baselines, we see rather similar performance in terms of cost in most cases. While using the most vital edge-based baseline often provides some improvement over cost alone, it never yields better than a 10% improvement over GreedyCost. This is true despite having considerably greater running time. It is important to note that our GreedyMVE implementation is not optimized: it is implemented in NetworkX and uses a brute-force technique to find the most vital edge (i.e., find the shortest path after removing each candidate edge from the graph). The cost results, however, suggest that even an optimized version would provide relatively little improvement over greedily removing the least costly edge.

Comparing the cost achieved by PATHATTACK to those obtained by the greedy baseline, we observe some interesting phenomena. Lattices and road networks, for example, have a similar tradeoff: PATHATTACK provides a mild improvement in cost at the expense of an order of magnitude additional processing time. Considering that PATHATTACK-Rand usually results in the optimal solution (over 86% of the time), this means that the baselines often achieve near-optimal cost with a naïve algorithm. On the other hand, ER, BA, and KR graphs follow a trend more similar to the AS and WIKI networks, particularly in the randomly weighted cases: the cost is cut by a substantial fraction—enabling the attack with a smaller budget—for a similar or smaller time increase. This suggests that the time/cost tradeoff is much less favorable for less clustered, grid-like networks (note the clustering coefficients in Appendix D).

Cliques (COMP) are particularly interesting in this case, showing a phase transition as the entropy of the weights increases. When edge weights are equal, cliques behave like an extreme version of the road networks: an order of magnitude increase in runtime with no decrease in cost. With Poisson weights, PATHATTACK yields a slight improvement in cost, whereas when uniform random weights are used, the clique behaves much more like an ER or BA graph. In the unweighted case,

[7] Gurobi is at https://www.gurobi.com. NetworkX is at https://networkx.org. Code from the experiments is at https://github.com/bamille1/PATHATTACK.

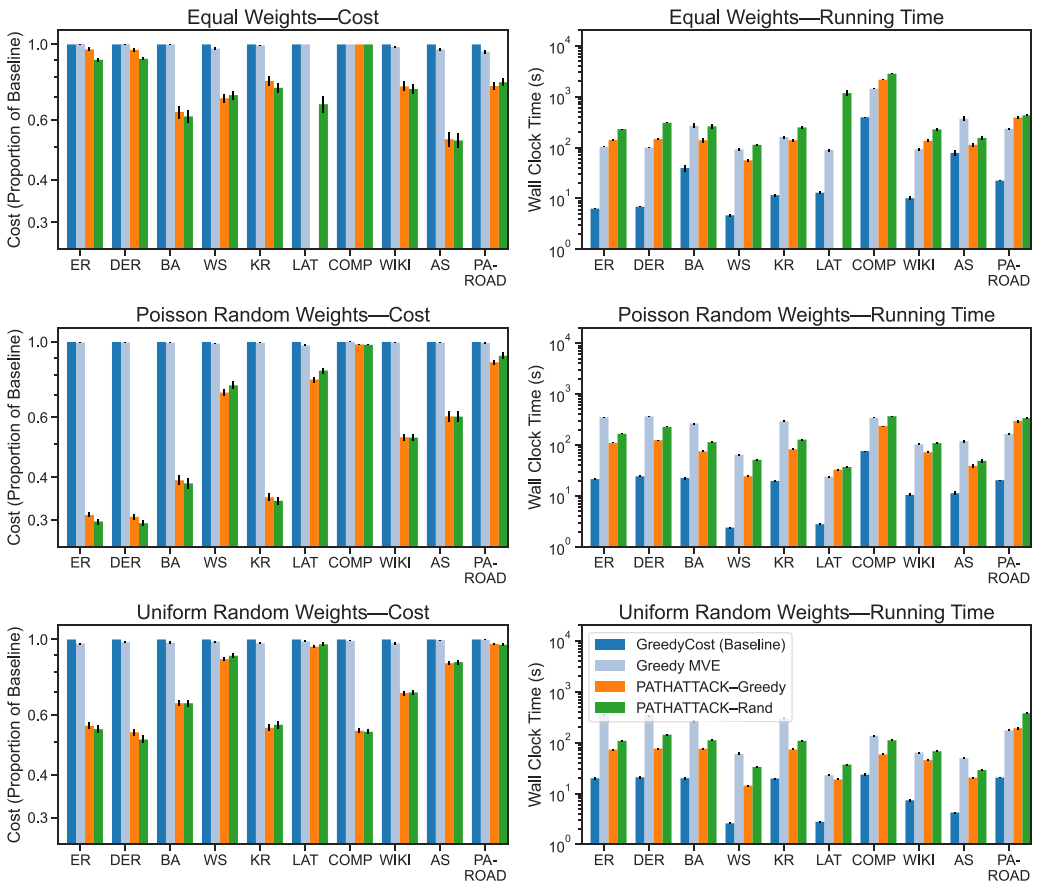

Fig. 4. PATHATTACK results on synthetic and real unweighted graphs with synthetic weights added, in terms of edge removal cost (left column) and running time (right column). Lower is better for both metrics. Cost is plotted as a proportion of the cost using the GreedyCost baseline. Bar heights are means across 100 trials and error bars are standard errors. Results are shown with equal weights on all edges (top row) and edge weights drawn from Poisson (middle row) and uniform (bottom row) distributions. PATHATTACK-Greedy operated on lattices with equal weights for over 1 day without converging, so results were not collected for this case. PATHATTACK yields a substantial reduction in cost in ER, BA, KR, WIKI, and AS graphs, while the baseline is often near optimal for LAT and PA-ROAD.

p^* is a three-hop path, so all other two- and three-hop paths from s to t must be cut, which the baseline does efficiently. Adding Poisson weights creates some randomness, but most edges have a weight that is about average, so it is still similar to the unweighted scenario. With uniform random weights, we get the potential for much different behavior (e.g., short paths with many edges) for which the greedy baseline's performance suffers.

There is an opposite, but milder, phenomenon with PA-ROAD and LAT: using higher-entropy weights *narrows* the cost difference between the baseline and PATHATTACK. This may be due to the source and destination being many hops away. With the terminal nodes many hops apart, many shortest paths between them could go through a few low-weight (thus low-cost) edges. A very low-weight edge between two nodes would be very likely to occur on many of the shortest paths, and would be found in an early iteration of the greedy algorithm and removed, while considering

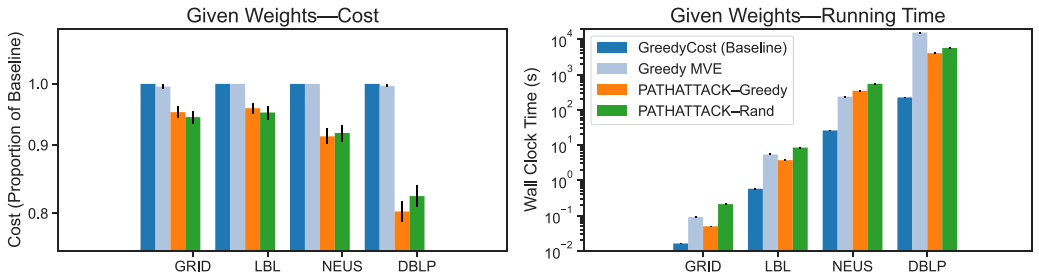

Fig. 5. PATHATTACK results on real weighted graphs in terms of edge removal cost (left) and running time (right). Lower is better for both metrics. Cost is plotted as a proportion of the cost using the GreedyCost baseline. Bar heights are means across 100 trials and error bars are standard errors. Note the difference in scale on the vertical axes from Figure 4. PATHATTACK yields a substantial improvement in performance for DBLP, while the baseline performs well on the other (less clustered) networks.

more shortest paths at once would yield a similar result. We also note that, in the weighted graph data, LBL and GRID behave similarly to road networks. Among our real datasets, these have a low clustering coefficient (see Appendix D). This lack of overlap in nodes' neighborhoods may lead to better relative performance with the baseline, since there may not be a great deal of overlap between candidate paths. With the real datasets, we also see near-optimal performance with the baseline method. Take the LBL dataset for example. In 60 out of 100 trials aggregated for this dataset in Figure 5, PATHATTACK-Rand yields the optimal solution. Among these trials, the GreedyCost baseline yields, on average, about a 9.3% increase in cost. This is very similar to the 7.1% increase in cost that the baseline yields among *all* trials. This demonstrates that, in many real-world networks, greedy methods are highly effective.

6.5 Results Targeting a Node

In this section, we present results for the case where the adversary targets a node v^*. The baseline in these experiments is to run PATHATTACK using the shortest path from s to t through v^* as p^*. We call this method PATHATTACK-v^*. We present results using this method along with results for the heuristic search method (Algorithm 6) and combinatorial optimization.

In most cases, all three methods yield a solution of the same cost. To clarify the performance differences, we separate these cases from those where the costs differ between methods. For the case where all methods result in the same cost, we show running time results on unweighted graphs in Figure 6. As when targeting paths, lattices and road networks are similar: they are the only graphs where the heuristic methods do not match the result of the combinatorial optimization a majority of the time, largely due to the equal-weight case. Watts–Strogtaz graphs, which have a lattice-like component, also frequently have different results across methods. Figure 7 illustrates cases where not all algorithms yield the same cost. Again, lattices and roads are distinct: here they see a much more substantial improvement from the combinatorial optimization than heuristics, with Watts–Strogatz graphs also sometimes being similar. There are many cases where the heuristic search yields the same result as the PATHATTACK-v^* baseline. Upon inspection, many of these cases result from multiple shortest paths using v^*: the heuristic overlooks one solution because it does not cut the current p^*, which results in no edges to consider in the inner loop of Algorithm 6.

For weighted networks, running time in cases where all methods yield the same cost is plotted in Figure 8. In all cases, the three algorithms usually find equal-cost solutions, though the combinatorial method frequently times out. The power grid network has a particularly large increase in running time when using the combinatorial method. In cases where not all methods yield the same

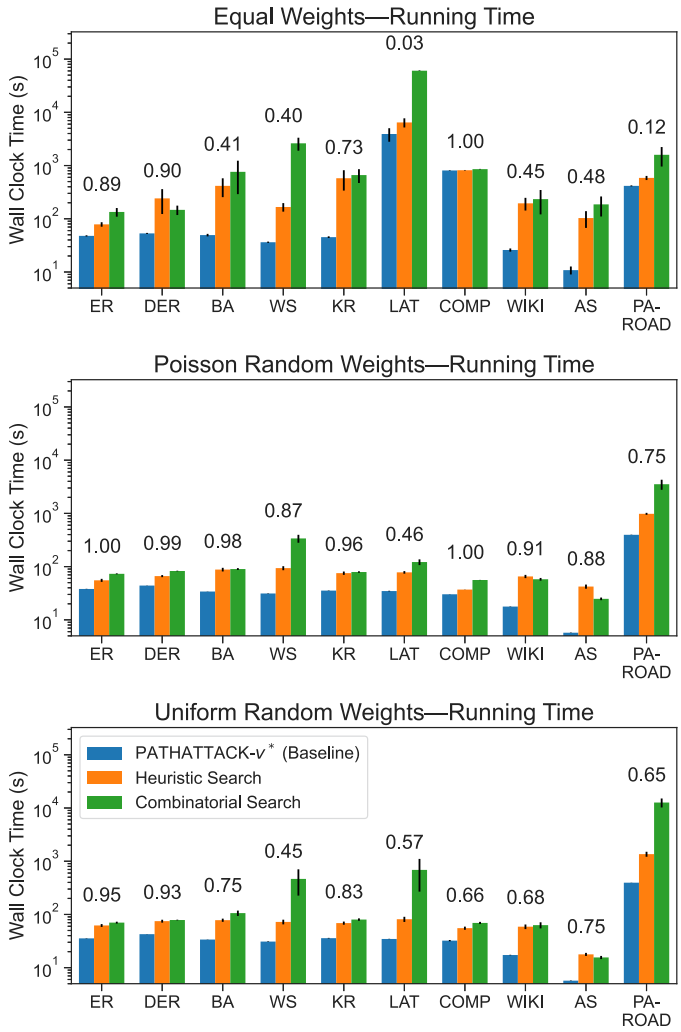

Fig. 6. Running time results on unweighted networks when targeting a specific node in cases where all three algorithms yield the same cost. Lower time is better. Annotations are the proportion of 100 trials where all costs are the same. Bar heights are means across these trials and error bars are standard errors. Results are shown with equal weights on all edges (top) and edge weights drawn from Poisson (middle) and uniform (bottom) distributions. With the exception of LAT and PA-ROAD, the baseline method matches the combinatorial optimization in a majority of cases and has a substantially smaller time requirement.

cost, shown in Figure 9, the heuristic search achieves the same cost as the combinatorial search more often than in the unweighted graphs. In fact, on DBLP, the combinatorial search typically times out and the heuristic search outperforms it.

7 CONCLUSION

In this article, we introduce the Force Path Cut, Force Edge Cut, and Force Node Cut problems, in which edges are removed from a graph in order to make the shortest path from one node to another, respectively, be a specific path, use a specific edge, or use a specific node. We show that

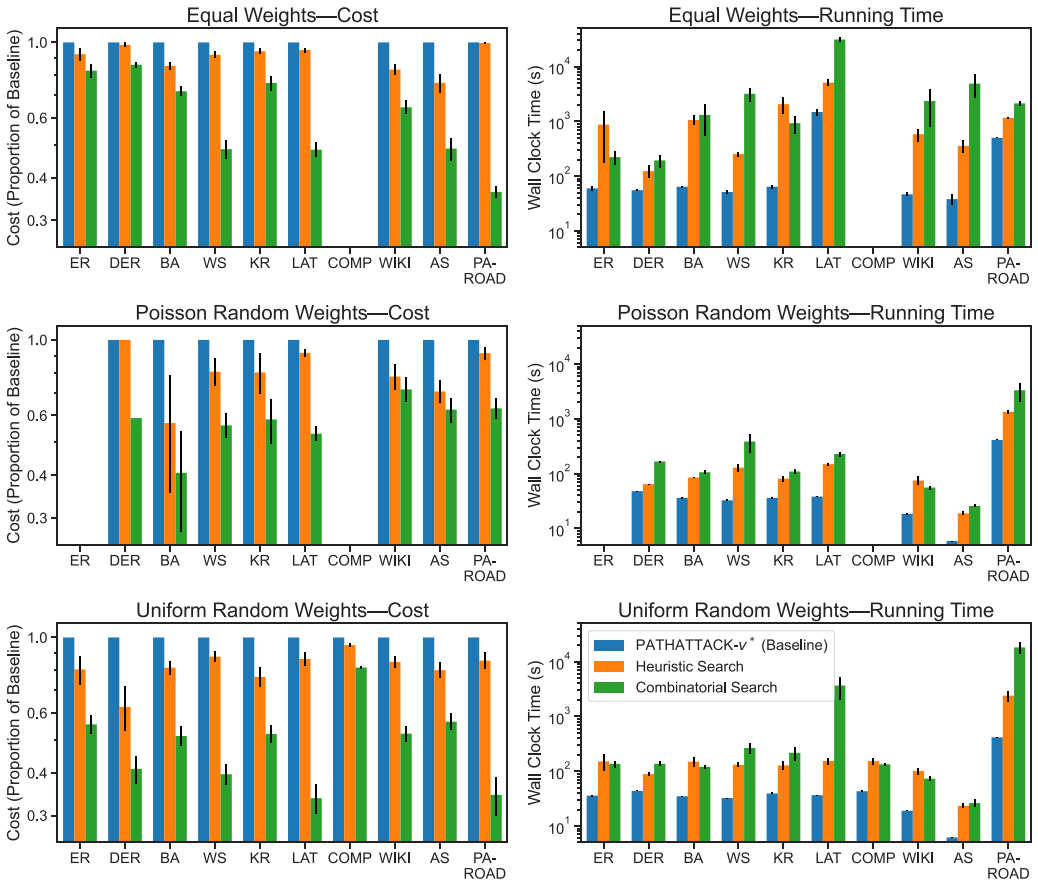

Fig. 7. Results on unweighted networks when targeting a specific node in cases where not all node-targeting algorithms yield the same cost, in terms of edge removal cost (left column) and running time (right column). Lower is better for both metrics. Bar heights are means across these trials and error bars are standard errors. Results are shown with equal weights on all edges (top row) and edge weights drawn from Poisson (middle row) and uniform (bottom row) distributions. Note that COMP with equal and Poisson weights and ER with Poisson weights are not included, as the methods matched in all trials. The greatest cost reductions from using combinatorial search often coincide with large running time increases.

the optimization versions of all three problems are hard to approximate, but that a logarithmic approximation exists for Optimal Force Path Cut via existing approximation algorithms for Set Cover. We leverage these methods to develop a new algorithm called PATHATTACK and demonstrate its efficacy in solving Optimal Force Path Cut using a thorough set of experiments on real and synthetic networks. We also use PATHATTACK as part of a heuristic search method to solve Optimal Force Edge Cut and Optimal Force Node Cut, which yields performance similar to a much more computationally intensive combinatorial search.

There is a gap between the approximation factor of PATHATTACK and the lower bound implied by APX-hardness; it remains an open problem whether Optimal Force Path Cut is APX-complete or if there is no constant-factor approximation. The best possible polynomial-time approximation factor for 3-Terminal Cut is 12/11 [17], and our results imply that the best such approximation for Optimal Force Path Cut is between 12/11 and logarithmic. Other approximation algorithms,

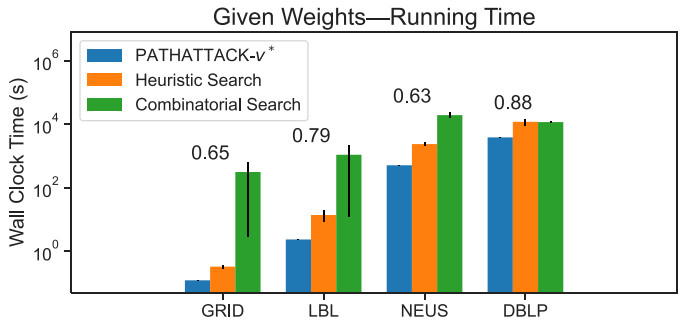

Fig. 8. Running time results on weighted networks when targeting a specific node in cases where all three algorithms yield the same cost. Lower time is better. Annotations are the proportion of 100 trials where all costs are the same. Bar heights are means across these trials and error bars are standard errors. In contrast to the unweighted networks, the heuristic methods match the combinatorial optimization in a large majority of cases, even in grid-like networks.

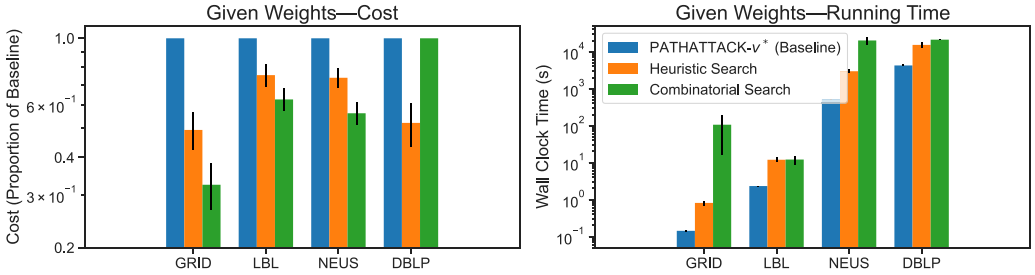

Fig. 9. Results on weighted networks when targeting a specific node in cases where not all node-targeting algorithms yield the same cost, in terms of edge removal cost (left) and running time (right). Lower is better for both metrics. Bar heights are means across these trials and error bars are standard errors. Heuristic search in these cases is competitive with combinatorial search, even outperforming it sometimes due to timeouts.

such as those for hypergraph vertex cover [31], may be useful in some scenarios, such as very small-diameter graphs. In addition, while we provide upper bounds for the running times of all algorithms, the experimental results suggest these are pessimistic and could be tightened. In particular, in the cases where we solve sparse linear programs, the structure of the sparse matrices could potentially be exploitable to achieve tighter bounds. Future work will include defenses against PATHATTACK to make networks more robust against adversarial manipulation.

APPENDICES

A PROOF OF THEOREM 3.1

A.1 Proof for Undirected Graphs

As noted in Section 3.1, to prove Theorem 3.1, we first prove that Optimal Force Path Cut is APX-hard for undirected graphs.

Lemma A.1. *Optimal Force Path Cut is APX-hard for undirected graphs, including the case where all weights and all costs are equal.*

To prove Lemma A.1, we reduce 3-Terminal Cut to Force Path Cut via a linear reduction. Let $G = (V, E)$ be an undirected graph, where all weights are equal. We also have three terminal nodes

s_1, s_2, and $s_3 \in V$. Since we are proving the problem is APX-hard, we consider the optimization versions of both problems, where the goal is to minimize the budget. Thus, the goal of 3-Terminal Cut is to find the smallest set E' such that s_1, s_2, and s_3 are disconnected in $G' = (V, E \backslash E')$. Dahlhaus et al. show in [18] that 3-Terminal Cut is APX-hard, even when all weights are equal.[8]

We propose a linear reduction from 3-Terminal Cut to Force Path Cut. As discussed in [18], a linear reduction from problem A to problem B consists of two functions f and g, where f maps an instance of A to an instance of B, and g maps a solution of B to a solution of A. To be a linear reduction, the following conditions must hold:

(1) The functions f and g can be computed in polynomial time.
(2) Given $\mathcal{A}$, an instance of problem A, the optimal solution to $\mathcal{B} = f(\mathcal{A})$ must be at most α times the optimal solution for $\mathcal{A}$, for a constant $\alpha > 0$, i.e., $\text{opt}(\mathcal{B}) \leq \alpha \cdot \text{opt}(\mathcal{A})$.
(3) Given a solution y to $\mathcal{B} = f(\mathcal{A})$, $x = g(y)$ is a solution to $\mathcal{A}$ such that

$$|\text{cost}(x) - \text{opt}(\mathcal{A})| \leq \beta |\text{cost}(y) - \text{opt}(\mathcal{B})|$$

for a constant $\beta > 0$.

We start by defining f, the function from an instance of 3-Terminal Cut to an instance of Optimal Force Path Cut. We are given an instance of 3-Terminal Cut as described above. Let $N = |V|$ and $M = |E|$. As shown in Figure 1, we add $M + 1$ new paths of length N from s_1 to s_2, and the same from s_2 to s_3. We add a single path of length $2N - 1$ from s_1 to s_3 and make this path p^*. Algorithm 8 provides pseudocode for this procedure.

Applying Algorithm 8 to an optimization instance of 3-Terminal Cut (G, s_1, s_2, s_3), we get (G', s_1, s_3, p^*), an instance of Optimal Force Path Cut. Note that, in these instances, all edge weights and removal costs are equal to 1. From the instance of Optimal Force Path Cut, we get a solution E' consisting of edges whose removal results in p^* being the shortest path from s_1 to s_3. We also have a function that maps a solution to Optimal Force Path Cut to a solution to 3-Terminal Cut: include all edges in E' that existed in the original graph, i.e.,

$$g(E') = \begin{cases} E' \cap E & \text{if } |E'| < M \\ E & \text{otherwise,} \end{cases} \tag{29}$$

where E' is the solution to Optimal Force Path Cut and E is the original edge set from the 3-Terminal Cut instance. (The edge set E is not a parameter of g, as it is fixed within the context of the problem.)

We can see that both functions satisfy condition (1): the function g simply removes up to M edges from a set, and the body of each loop takes constant time in Algorithm 8, and there are two nested loops taking $O(MN)$ time and a final loop taking $O(N)$ time. To show that this reduction satisfies condition (2), we first prove the following Lemma.

Lemma A.2. *Let $\mathcal{A}$ be an instance of 3-Terminal Cut and E' be a solution to $\mathcal{A}$. Then E' is also a solution to $f(\mathcal{A})$.*

Proof. Since E' is a solution to $\mathcal{A}$, the graph $G' = (V, E \setminus E')$ has at least three connected components, where one contains s_1, one contains s_2, and one contains s_3. The edges E_{new} added by Algorithm 8 (in the creation of $\hat{G}$) create paths between the connected components but do not connect to any vertices in the original graph other than s_1, s_2, and s_3. Thus, there are two modes of traversing from s_1 to s_3 via a simple path: (1) traverse the new path from s_1 to s_3 denoted as p^* by Algorithm 8, or (2) move from s_1 to s_2 via edges from E_{new}, then from s_2 to s_3 via edges from E_{new}.

[8] More specifically, it is proved in [18] that the problem is MAX SNP-hard, but this implies APX-hardness: if a problem is MAX SNP-hard, it has no polynomial-time approximation scheme unless P = NP.

ALGORITHM 8: Mapping from 3-Terminal Cut to Optimal Force Path Cut.

Input: Graph $G = (V, E)$, terminals s_1, s_2, s_3
Output: Graph $\hat{G}$, target path p^*
$\hat{V} \leftarrow V$; $N \leftarrow |V|$;
$\hat{E} \leftarrow E$; $M \leftarrow |E|$;
// Create paths from s_1 to s_2
for $i \leftarrow 1$ *to* $M + 1$ **do**
  $v_{\text{prev}} \leftarrow s_1$;
  for $j \leftarrow 1$ *to* $N - 1$ **do**
    $v_{ij,1} \leftarrow$ new node;
    $\hat{V} \leftarrow \hat{V} \cup \{v_{ij,1}\}$;
    $\hat{E} \leftarrow \hat{E} \cup \{\{v_{\text{prev}}, v_{ij,1}\}\}$;
    $v_{\text{prev}} \leftarrow v_{ij,1}$;
  end
  $\hat{E} \leftarrow \hat{E} \cup \{\{v_{\text{prev}}, s_2\}\}$;
end
// Create paths from s_2 to s_3
for $i \leftarrow 1$ *to* $M + 1$ **do**
  $v_{\text{prev}} \leftarrow s_2$;
  for $j \leftarrow 1$ *to* $N - 1$ **do**
    $v_{ij,2} \leftarrow$ new node;
    $\hat{V} \leftarrow \hat{V} \cup \{v_{ij,2}\}$;
    $\hat{E} \leftarrow \hat{E} \cup \{\{v_{\text{prev}}, v_{ij,2}\}\}$;
    $v_{\text{prev}} \leftarrow v_{ij,2}$;
  end
  $\hat{E} \leftarrow \hat{E} \cup \{\{v_{\text{prev}}, s_3\}\}$;
end
// Create p^* (a path from s_1 to s_3)
$v_{\text{prev}} \leftarrow s_1$;
$p^* \leftarrow$ empty path;
for $j \leftarrow 1$ *to* $2N - 2$ **do**
  $v_{ij,3} \leftarrow$ new node;
  $\hat{V} \leftarrow \hat{V} \cup \{v_{ij,3}\}$;
  $\hat{E} \leftarrow \hat{E} \cup \{\{v_{\text{prev}}, v_{ij,3}\}\}$;
  append $\{v_{\text{prev}}, v_{ij,3}\}$ to the end of p^*;
  $v_{\text{prev}} \leftarrow v_{ij,3}$;
end
$\hat{E} \leftarrow \hat{E} \cup \{\{v_{\text{prev}}, s_3\}\}$;
append $\{v_{\text{prev}}, s_3\}$ to the end of p^*;
return $\hat{G} = (\hat{V}, \hat{E})$, s_1, s_3, p^*;

By construction, the path directly from s_1 to s_3 (the path added in the final loop of Algorithm 8) passes through $2N - 2$ intermediate nodes, having a length of $2N - 1$. Taking the indirect route first requires taking one of the $M + 1$ paths from s_1 to s_2, which has length N, then taking one of the $M + 1$ paths from s_2 to s_3, which also has length N. Thus, the total length of any path via s_2 is $2N$, which is longer than p^*. Thus, p^* is the shortest path from s_1 to s_3 in $\hat{G}' = (\hat{V}, \hat{E} \setminus E')$, so E' is a solution to $f(a)$. □

An immediate consequence of Lemma A.2 is that condition (2) is satisfied, as stated formally below.

Corollary A.3. *If $\mathcal{A}$ is an instance of 3-Terminal Cut, then* $\text{opt}(f(\mathcal{A})) \leq \text{opt}(\mathcal{A})$*, satisfying condition (2) with* $\alpha = 1$.

Proof. Let E' be the optimal solution to $\mathcal{A}$, i.e., $|E'| = \text{opt}(\mathcal{A})$. By Lemma A.2, E' also solves $f(\mathcal{A})$. Thus, the optimal solution to $f(\mathcal{A})$ can be no larger than $|E'|$, and therefore $\text{opt}(f(\mathcal{A})) \leq \text{opt}(\mathcal{A})$. □

While the above corollary is sufficient to satisfy condition (2), we can make a stronger statement that is useful to prove condition (3): the optimal solutions of the two problems are the same.

Lemma A.4. *For an instance of 3-Terminal Cut $\mathcal{A}$,* $\text{opt}(\mathcal{A}) = \text{opt}(f(\mathcal{A}))$*. In particular, if E' is an optimal solution for $\mathcal{A}$, then it is also an optimal solution for $f(\mathcal{A})$.*

Proof. Let $G = (V, E)$ be the graph in $\mathcal{A}$, and $\hat{G} = (\hat{V}, \hat{E})$ be the graph in problem $f(\mathcal{A})$. Let $\hat{E}'$ be an optimal solution to $f(\mathcal{A})$. Partition $\hat{E}'$ into the edges that occur in the original graph $E_1 = \hat{E}' \cap E$, and those that do not, $E_2 = \hat{E}' \setminus E$. By Lemma A.2, if E_1 is a solution to $\mathcal{A}$, it is also a solution to $f(\mathcal{A})$. Therefore, if E_1 is a solution to $\mathcal{A}$, $E_2 = \emptyset$. (Otherwise E_1 is a solution to $f(\mathcal{A})$ and $|E_1| < |\hat{E}'|$, which contradicts the assumption that $\hat{E}'$ is an optimal solution to $f(\mathcal{A})$.) Thus, we focus on the case where E_1 is not a solution to $\mathcal{A}$. In this case, within the graph $G_1 = (V, E \setminus E_1)$, not all terminals s_1, s_2, and s_3 are disconnected. If there is a path from s_1 to s_3, the length of this path is at most $N-1$, which is shorter than p^*. This contradicts the assumption that $\hat{E}'$ is a solution to $f(\mathcal{A})$—E_2 only includes edges not in the original graph, so the shorter path will still exist when edges from E_2 are removed. There are two other possibilities: s_1 and s_2 are connected in G_1, or s_2 and s_3 are connected. If s_1 and s_2 are connected, there is a path between the terminals of length at most $N-2$ (excluding s_3). Algorithm 8 inserts $M+1$ independent parallel paths from s_2 to s_3 in $\hat{E}$. If any of these paths remains, there is a path in $\hat{G}' = (\hat{V}, \hat{E} \setminus \hat{E}')$ from s_2 to s_3 of length N, which would create a path from s_1 to s_3 of length at most $2N-2$, which is shorter than p^*. Thus, $\hat{E}'$ would have to include at least one edge from all $M+1$ parallel paths from s_2 to s_3 inserted by Algorithm 8. This means that $|E_2| \geq M+1$, implying that $|\hat{E}'| \geq M+1$. This contradicts the assumption that $\hat{E}'$ is an optimal solution to $f(\mathcal{A})$: any solution to $\mathcal{A}$ is a solution to $f(\mathcal{A})$ and its cost is at most M. The analogous argument holds if s_2 and s_3 are connected in G_1. Thus, a solution to $f(\mathcal{A})$ cannot be optimal if it does not include a solution to $\mathcal{A}$, implying that $\text{opt}(f(\mathcal{A})) \geq \text{opt}(\mathcal{A})$. This in conjunction with Corollary A.3 proves that the optima of $\mathcal{A}$ and $f(\mathcal{A})$ are the same. □

To show that the reduction also satisfies condition (3), we first prove the following lemma.

Lemma A.5. *Let $\mathcal{A}$ be an instance of 3-Terminal Cut where all weights are equal, with $G = (V, E)$. Let $\mathcal{B} = f(\mathcal{A})$ be the instance of Optimal Force Path Cut obtained by applying Algorithm 8 to $\mathcal{A}$, with $\hat{G} = (\hat{V}, \hat{E})$. Further, let $\hat{E}'$ be a solution to $\mathcal{B}$. Then $g(\hat{E}')$ is a solution to $\mathcal{A}$.*

Proof. The case where $|\hat{E}'| \geq M = |E|$ is trivial: if all edges are removed, then the terminals are disconnected. With the assumption that $|\hat{E}'| < M$, partition $\hat{E}'$ into two parts: $E_1 = \hat{E}' \cap E$ and $E_2 = \hat{E}' \setminus E1$; i.e., E_1 is the edges in the solution to $\mathcal{B}$ that existed in the original graph, and E_2 consists of the edges in the solution added by f (Algorithm 8). Since $\hat{E}'$ is a solution to $\mathcal{B}$, p^* is the shortest path from s_1 to s_3 in the graph $\hat{G}' = (\hat{V}, \hat{E} \setminus \hat{E}')$. The length of p^* is $2N-1$. Assume E_1 is not a solution to $\mathcal{A}$, i.e., that s_1, s_2, and s_3 are not disconnected in $G_1 = (V, E \setminus E_1)$. Then there would be a path between at least two of the three terminals in G_1. If there were a path from s_1 to s_3, the length of this path would be at most $N-1 < 2N-1$, so this contradicts the assumption

that p^* is the shortest path in $\hat{G}'$. For the other cases, assume there is no such path (i.e., s_1 and s_3 are disconnected in G_1). Suppose there is a path between s_1 and s_2. This path's length is at most $N-2$ (since it cannot include s_3). In addition, either (1) one of the paths from s_2 to s_3 added by f remains or (2) all $M+1$ such paths were cut, which contradicts our assumption since it would require $\hat{E}'$ to contain at least $M+1$ edges. The paths from s_2 to s_3 added in Algorithm 8 have length N, thus creating a path from s_1 to s_3 with length at most $2N-2 < 2N-1$, which also contradicts the assumption that p^* is the shortest path. A similar argument is made for the case where a path from s_2 to s_3 remains in G_1: such a path would have length at most $N-2$, and at least one of the paths added between s_1 and s_2 remains, resulting in a path of length at most $2N-2$. Thus, in the case where $|\hat{E}'| < M$, $\hat{E}' \cap E$ is a solution to $\mathcal{A}$. □

With this result, we can show that the reduction meets the final criterion.

Lemma A.6. *If $\mathcal{A}$ is an instance of 3-Terminal Cut with $G = (V, E)$, y is a solution to $\mathcal{B} = f(\mathcal{A})$, and $x = g(y)$, then*

$$|\text{cost}(x) - \text{opt}(\mathcal{A})| \leq |\text{cost}(y) - \text{opt}(\mathcal{B})|,$$

satisfying (3) with $\beta = 1$.

Proof. Let $\hat{E}'$ be a solution to $\mathcal{B}$, and partition $\hat{E}'$ into $E_1 = \hat{E}' \cap E$ and $E_2 = \hat{E}' \setminus E_1$. From Lemma A.5, we know that $g(\hat{E}')$ solves $\mathcal{A}$. From Lemma A.4, we also know the optimal solutions are the same size. If $|\hat{E}'| \geq M$, then $g(\hat{E}') = E$, and we have

$$||E| - \text{opt}(\mathcal{A})| = |M - \text{opt}(\mathcal{A})| = |M - \text{opt}(\mathcal{B})| \leq \left||\hat{E}'| - \text{opt}(\mathcal{B})\right|, \tag{30}$$

so the condition holds. If $|\hat{E}'| < M$, we have

$$\left||\hat{E}' \cap E| - \text{opt}(\mathcal{A})\right| = \left||\hat{E}' \cap E| - \text{opt}(\mathcal{B})\right| \leq \left||\hat{E}'| - \text{opt}(\mathcal{B})\right|, \tag{31}$$

which proves the claim. □

These intermediate results show that the proposed reduction is a linear reduction of 3-Terminal Cut to Force Path Cut, implying that Optimal Force Path Cut is APX-hard.

Proof of Lemma A.1. By construction, f, as described by Algorithm 8, takes an instance of 3-Terminal Cut and maps it to an instance of Force Path Cut. As shown in the pseudocode, this takes $O(MN)$ time to compute. By Lemma A.5, g maps a solution to the instance of Force Path Cut obtained via f to a solution to the original 3-Terminal Cut problem. The procedure of removing the original edge set takes polynomial time that varies depending on the data structure, e.g., $O(MN)$ per removal in an adjacency list. Thus, f and g are appropriate mappings that take polynomial time to compute, satisfying condition (1). By Corollary A.3, the reduction satisfies condition (2), and by Lemma A.6, it satisfies (3). This means that f and g provide a linear reduction from 3-Terminal Cut to Force Path Cut. Since optimizing 3-Terminal Cut is APX-hard, Optimal Force Path Cut for undirected graphs is APX-hard as well. □

A.2 Proof for Directed Graphs

To prove that Optimal Force Path Cut is APX-hard for directed graphs, we formulate a linear reduction from Force Path Cut for undirected graphs to the directed case. The linear reduction in this case is simple. The function f that maps an undirected instance of Force Path Cut to a directed one simply takes each edge from the former and includes the two directed edges between

the associated nodes (one in each direction). The values of p^*, s, and t remain the same. Formally, f replaces E with

$$\hat{E} = \bigcup_{\{u,v\} \in E} \{(u,v),(v,u)\}. \tag{32}$$

The graph $\hat{G} = (V, \hat{E})$ can be constructed in $O(N + M)$ time.

If we have a solution to Force Path Cut on the directed graph $\hat{G}$, there is a similarly simple mapping to a solution to undirected Force Path Cut: include the undirected version of each edge in the solution. This takes $O(M)$ time. We show that this mapping provides a solution to the original problem in the following lemma.

Lemma A.7. *Let $\mathcal{A}$ be an undirected instance of Force Path Cut, and $f(\mathcal{A})$ be its corresponding directed instance. If $\hat{E}'$ is a solution to $f(\mathcal{A})$, then the undirected solution*

$$E' = \bigcup_{(u,v) \in \hat{E}'} \{\{u,v\}\} \tag{33}$$

solves $\mathcal{A}$.

Proof. Suppose E' did not solve $\mathcal{A}$; i.e., p^* is not the shortest path from s to t in $G' = (V, E \setminus E')$. Then there is some other path, $\hat{p}$, from s to t that is not longer than p^*. However, this path also exists in $\hat{G}' = (V, \hat{E} \setminus \hat{E}')$: all edges from E were added in the creation of $\hat{E}$, so if $\hat{p}$ were not in $\hat{G}'$, at least one edge from $\hat{p}$ would have to be in $\hat{E}'$. The mapping g would include the undirected version of this edge in E', which would cut $\hat{p}$ in G as well. Thus, the existence of $\hat{p}$ contradicts the assumption that $\hat{E}'$ is a solution to $f(\mathcal{A})$, proving the claim. □

Lemma A.8. *Let E' be a solution to Force Path Cut on a directed graph. If $(u,v) \in E'$ and $(v,u) \in E'$, then either $E' \setminus \{(u,v)\}$ or $E' \setminus \{(v,u)\}$ is also a solution to Force Path Cut.*

Proof. Let $d(v_1, v_2)$ be the distance between v_1 and v_2 in the directed graph $G' = (V, E \setminus E')$. (If a path does not exist between v_1 and v_2, then $d(v_1, v_2) = \infty$.)

First, consider the case where both u and v are on p^*. Assume without loss of generality that u precedes v on the path. Then (v,u) did not need to be removed from the edge set. Since p^* is the shortest path from s to t, it is composed of shortest paths between all intermediate nodes on the path; for example, the shortest path from s to v and the shortest path from v to t. Removing (v,u) from E'—leaving the edge in the graph when solving Force Path Cut—would not change the status of p^* as the shortest path from s to t: the shortest path from s to v would still include u, and adding (v,u) back to the graph would only enable moving backward along the path. This means that $E' \setminus \{(v,u)\}$ is also a solution.

Now consider a case where one of u and v is part of p^*, but the other is not. Without loss of generality, assume u is on p^* and v is not. Let ℓ_{p^*} be the length of p^*. Since p^* is the shortest path, we know that $d(s,v) + d(v,t) > \ell_{p^*}$. Since u is on p^*, we have $d(s,u) + d(u,t) = \ell_{p^*}$. Suppose the claim does not hold: that neither $E' \setminus \{(u,v)\}$ nor $E' \setminus \{(v,u)\}$ is a solution. This means that adding (u,v) back into the edge set must create a path not longer than p^*, so we have $d(s,u) + d(v,t) + 1 \leq \ell_{p^*}$. The same is true for (v,u), implying that $d(s,v) + d(u,t) + 1 \leq \ell_{p^*}$. Adding the latter two inequalities, we have

$$d(s,u) + d(u,t) + d(s,v) + d(v,t) + 2 \leq 2\ell_{p^*} \Rightarrow d(s,v) + d(v,t) \leq \ell_{p^*} - 2, \tag{34}$$

where we use the equation $d(s,u) + d(u,t) = \ell_{p^*}$. This contradicts the first inequality, that the path from s to t via v is longer than p^*.

In the case where neither u nor v is on p^*, we replace $d(s,u)+d(u,t) = \ell_{p^*}$ with $d(s,u)+d(u,t) > \ell_{p^*}$, and the inequality in Equation (34) becomes strict. Thus, if both (u,v) and (v,u) are in E', then either $E' \setminus \{(u,v)\}$ or $E' \setminus \{(v,u)\}$ is also a solution to Force Path Cut. □

This lemma has an immediate consequence that is important for proving the reduction is linear.

Corollary A.9. *Let E' be an optimal solution to Force Path Cut on a directed graph. If $(u,v) \in E'$, then $(v,u) \notin E'$.*

Proof. From Lemma A.8, if E' contained both (u,v) and (v,u), then removal of one of these edges would still be a solution. Since the resulting solution would be smaller than E', this contradicts the premise of the claim. □

In addition, we obtain the optimal solution to the undirected problem if we find the optimal solution to the directed problem via the reduction.

Lemma A.10. *Let $\mathcal{A}$ be an undirected instance of Force Path Cut and $f(\mathcal{A})$ be the corresponding directed instance. Then the optimal solution to $f(\mathcal{A})$ is the optimal solution to $\mathcal{A}$.*

Proof. By Lemma A.7, we know that the optimal solution to $f(\mathcal{A})$ solves $\mathcal{A}$, so $\text{opt}(\mathcal{A}) \leq \text{opt}(f(\mathcal{A}))$. Let E' be an optimal solution to $\mathcal{A}$. Let $\hat{E}'$ be a solution to $f(\mathcal{A})$ that includes both directed edges for each undirected edge in E', i.e.,

$$\hat{E}' = \bigcup_{\{u,v\} \in E'} \{(u,v),(v,u)\}. \tag{35}$$

Since E' is a solution to $\mathcal{A}$, $\hat{E}'$ is a solution to $f(\mathcal{A})$. (Otherwise a path $\hat{p}$ in $f(\mathcal{A})$ that is shorter than p^* would not be removed by E'.) However, $\hat{E}'$ contains a pair of edges for each edge in E': for any $(u,v) \in \hat{E}'$, we have $(v,u) \in \hat{E}'$. By Lemma A.8, one edge from each pair can be removed. This means that, for an optimal solution to $\mathcal{A}$, we can find a solution to $f(\mathcal{A})$ that is the same size, which implies that $\text{opt}(f(\mathcal{A})) \leq \text{opt}(\mathcal{A})$. This means that $\text{opt}(\mathcal{A}) = \text{opt}(f(\mathcal{A}))$, and an optimal solution for one problem can be applied to the other. □

Combining these intermediate results, we show that the reduction from undirected Force Path Cut to the directed version is linear, and the directed version of the optimization problem is also APX-hard.

Lemma A.11. *Optimal Force Path Cut is APX-hard for directed graphs, including the case where all weights and all costs are equal.*

Proof. The function f simply takes edges from an undirected graph and builds a directed graph with the same (directed) edges, which takes $O(N+M)$ time. The function g takes a set of directed edges and converts it into a set of undirected edges, which takes $O(M)$ time. By Lemma A.7, g maps to a true solution to the undirected Force Path Cut problem. This means that condition (1) is satisfied.

Lemma A.10 guarantees that condition (2) is satisfied as well, with $\alpha = 1$. Finally, let $\mathcal{A}$ be an undirected instance of Force Path Cut. For any solution $\hat{E}'$ to $f(\mathcal{A})$, we know that

$$\frac{1}{2}|\hat{E}'| \leq |g(\hat{E}')| \leq |\hat{E}'|. \tag{36}$$

Thus, we have

$$\left||g(\hat{E}')| - \text{opt}(\mathcal{A})\right| \leq \left||\hat{E}'| - \text{opt}(\mathcal{A})\right| = \left||\hat{E}'| - \text{opt}(\mathcal{B})\right|, \tag{37}$$

so condition (3) is satisfied with $\beta = 1$.

Since all three conditions are satisfied, f and g provide a linear reduction from Force Path Cut on an undirected graph to the same problem on a directed graph. By Lemma A.1, Optimal Force Path Cut for undirected graphs is APX-hard, so the reduction implies it is APX-hard for directed graphs as well. □

Theorem 3.1 is a direct consequence of Lemma A.1 and Lemma A.11.

B PATHATTACK CONVERGENCE

We consider the case where the ellipsoid algorithm is used to optimize the relaxed version of the integer program. At each iteration, we consider the center of an ellipsoid and determine whether this point violates any constraints. Call this point $\mathbf{x}^f_{\text{cut}} \in [0,1]^M$. In addition, let P be the current set of explicit path constraints and P_f be the set of path constraints—both implicit and explicit—that $\mathbf{x}^f_{\text{cut}}$ does not violate.

Given $\mathbf{x}^f_{\text{cut}}$, we perform the randomized rounding routine used in Algorithm 2. With probability at least 1/2, this procedure will yield a result that is within the guaranteed approximation margin (i.e., the objective of the integer solution is within $\ln(4|P|)$ of the fractional solution) and satisfying all explicit constraints. Thus, with probability at least $1 - \frac{1}{|E|}$, it will yield such a solution in $O(\log|E|)$ trials. More specifically, the probability of failing all of $\log_2|E|$ randomized rounding trials is bounded by

$$p_{\text{fail}} \leq \left(\frac{1}{2}\right)^{\log_2|E|} = \frac{1}{|E|}.$$

Note that this holds for the full set P_f if $|P_f| \leq \frac{1}{4}e^{\lceil \ln(4|P|)\rceil}$, since the two set sizes result in the same success bounds. If we attempt for $O(\log|E|)$ trials and never find a violated constraint, we increment the number of Bernoulli random variables used in the randomized rounding procedure and the approximation factor. With probability at least $1 - \frac{1}{|E|}$, this will yield a valid solution in $O(\log|E|)$ trials if $|P_f| \leq e^{\lceil \ln(4|P|)\rceil+1}$. We continue until we find a path that needs to be cut that is not (fractionally) cut by the solution of the relaxed problem or we obtain a solution that satisfies all constraints, implicit and explicit. Algorithm 9 provides pseudocode for this procedure. The algorithm will complete in polynomial time: There are at most $N \ln N$ iterations of the outer loop, at most $\lceil \log_2|E| \rceil$ iterations of the second loop, inside of which:

- At most $O(|E|N \log N)$ Bernoulli random variables are instantiated and aggregated.
- An $O(|E|)$-length vector is created.
- Edges are removed from a graph (at most $O(|E|)$ time).
- In the worst case, the *second* shortest path is found, which takes $O(N^3)$ time.
- The constraints are checked ($O(N)$ time).

Each item takes polynomial time to complete, so the overall algorithm takes polynomial time.

C PATHATTACK FOR NODE REMOVAL

As discussed in Section 4.5, PATHATTACK can be used for node removal in addition to edge removal. We refer to the associated optimization problem as *Optimal Force Path Remove*: given a weighted graph $G = (V, E)$, $w : E \to \mathbb{R}_{\geq 0}$, where each node has a cost of removal, $c : V \to \mathbb{R}_{\geq 0}$, a path p^* from $s \in V$ to $t \in V$, and a budget b, is there a subset of nodes $V' \subset V$ such that $\sum_{v \in V'} c(v) \leq b$ and p^* is the shortest path from s to t in $G' = (V \setminus V', E \setminus E_{V'})$, where $E_{V'} \subset E$ is the set of edges with at least one node in V'? In this section, we prove that this problem is also APX-hard and provide empirical results on the same datasets as shown in Section 6.

ALGORITHM 9: Randomized Rounding Oracle: apply randomized rounding to a candidate fractional solution, iteratively increasing the assumed number of implicit constraints if the procedure fails after $\log_2 |E|$ trials.

Input: Graph $G = (V, E)$, weights w, costs $\mathbf{c}$, path p^*, fractional cut vector $\mathbf{x}^f_{\text{cut}}$, path set P
Output: Path p that violates implicit constraints
$s \leftarrow$ first node of p^*;
$t \leftarrow$ last node of p^*;
apxFactor $\leftarrow \lceil \ln(4|P|) \rceil$;
$\hat{\mathbf{x}}_{\text{cut}} \leftarrow \mathbf{0}$;
$E' \leftarrow \emptyset$;
not_cut$\leftarrow$ **True**;
while (apxFactor $< N \ln N$) **and** not_cut **do**
  ctr $\leftarrow 0$;
  while ($\mathbf{c}^\top \mathbf{x}^f_{cut} > \mathbf{c}^\top \hat{\mathbf{x}}_{\text{cut}}(4 \cdot \text{apxFactor})$ **or** not_cut) **and** ctr $< \lceil \log_2 |E| \rceil$ **do**
    $E' \leftarrow \emptyset$;
    for $i \leftarrow 1$ *to* apxFactor **do**
      // randomly select edges based on $\mathbf{x}^f_{\text{cut}}$
      $E_1 \leftarrow \{e \in E$ with probability $\mathbf{x}^f_{\text{cut}}[e]\}$;
      $E' \leftarrow E' \cup E_1$;
    end
    $\hat{\mathbf{x}}_{\text{cut}} \leftarrow$ indicator vector for E';
    $G' \leftarrow (V, E \setminus E')$;
    $p \leftarrow$ shortest path (other than p^*) from s to t in G' with weights w;
    if $\sum_{(u,v)\in p} \mathbf{x}^f_{cut}[u, v] \geq 1$ **then**
      // p does not violate $\mathbf{x}^f_{\text{cut}}$
      not_cut$\leftarrow$**True**;
    else
      not_cut$\leftarrow$**False**;
      if p *is longer than* p^* **then**
        // found solution (if within approximation factor)
        $p \leftarrow \emptyset$;
      end
    end
    ctr $\leftarrow$ ctr + 1;
  end
  apxFactor $\leftarrow$ apxFactor + 1
end
return p

C.1 Computational Complexity

We propose the following reduction from Optimal Force Path Cut to Optimal Force Path Remove for unweighted, undirected graphs:

- Take the input graph $G = (V, E)$ and create its line graph $\hat{G} = (\hat{V}, \hat{E})$.
- Create new nodes $\hat{s}$ and $\hat{t}$ and add these to $\hat{V}$.
- For all edges $e \in E$ incident to s, create a new edge in $\hat{E}$ connecting $\hat{s}$ to the corresponding node $v_e \in \hat{V}$.

- For all edges $e \in E$ incident to t, create a new edge in $\hat{E}$ connecting $\hat{t}$ to the corresponding node $v_e \in \hat{V}$.
- Let $\hat{p}^*$ be the path from $\hat{s}$ to $\hat{t}$ where all intermediate nodes are those that correspond to the edges in p^*, in the same order.
- Solve Optimal Force Path Remove on $\hat{G}$ targeting $\hat{p}^*$.

This procedure is the function f that maps an instance of Optimal Force Path Cut to an instance of Optimal Force Path Remove. When we solve Optimal Force Path Remove, we obtain a set of nodes $\hat{V}'$ to remove. Since all nodes in $\hat{G}$ other than $\hat{s}$ and $\hat{t}$ correspond to edges in E—and $\hat{s}$ and $\hat{t}$ cannot be in $\hat{V}'$—the function g mapping a solution to Optimal Force Path Remove to a solution to Optimal Force Path Cut is to make E' the set of edges in E corresponding to the nodes in $\hat{V}'$.

We first prove that the reduction yields a solution to Optimal Force Path Cut in the following lemma.

Lemma C.1. *Let $\mathcal{A}$ be an unweighted, undirected instance of Optimal Force Path Cut and $\hat{V}'$ be a solution to $f(\mathcal{A})$. Then $E' = g(\hat{V}')$ is a solution to $\mathcal{A}$.*

Proof. Suppose E' were not a solution to $\mathcal{A}$. This means that p^* is not the shortest path from s to t in $G' = (V, E \setminus E')$. Let p' be the competing path. Since p' remains a path from s to t, none of its edges are in E', and therefore none of those edges' corresponding nodes from $\hat{G}$ are in $\hat{V}'$. This means that all edges in p' have nodes in $\hat{G}$. These nodes, however, form a path $\hat{p}'$ from $\hat{s}$ to $\hat{t}$, and since the length of p' is less than or equal to the length of p^*, $\hat{p}'$ is also no longer than $\hat{p}^*$, which contradicts the assumption that removing $\hat{V}'$ makes $\hat{p}^*$ the shortest path from $\hat{s}$ to $\hat{t}$. □

In addition, the optimal solution to Force Path Cut is also the optimal solution to the derived Force Path Remove instance.

Lemma C.2. *Let $\mathcal{A}$ be an instance of Optimal Force Path Cut. If $\hat{V}'$ is an optimal solution of $f(\mathcal{A})$, then $E' = g(\hat{V}')$ is an optimal solution of $\mathcal{A}$.*

Proof. Suppose there were a better solution $E_1 \subset E$, $|E_1| < |E'|$. Consider the corresponding set of nodes $V_1 \subset \hat{V}$. The augmented line graph of $G_1 = (V, E \setminus E_1)$ would be the same as $\hat{G}$ with V_1 (and all edges adjacent to nodes in V_1) removed. Since p^* is the shortest path from s to t in G_1, however, $\hat{p}^*$ would be the shortest path from $\hat{s}$ to $\hat{t}$ in the augmented line graph. This means that V_1 is a solution to $f(\mathcal{A})$. The fact that $|E_1| < |E'|$, however, implies that $|V_1| < |\hat{V}'|$, which contradicts the assumption that $\hat{V}'$ is an optimal solution, proving the claim. □

Using these results, we now prove APX-hardness in the undirected case.

Lemma C.3. *Optimal Force Path Remove is APX-hard for undirected graphs, including the case where all costs are equal.*

Proof. To prove the claim, we show that the reduction described above satisfies the requirements of a linear reduction: f and g can be computed in polynomial time, the optimal solutions differ by at most a constant factor, and the absolute difference of any solution from the optimal solution is bounded by a constant factor. We consider each criterion in turn.

To create the line graph, a new graph is created with $|E|$ nodes. For each node in the original graph, edges are created in the new graph connecting two of the incident edges. Thus, for a node in V with degree d, we have $\binom{d}{2}$ edges in $\hat{E}$, meaning that $|\hat{E}|$ is $O(|E|^2)$. After creating the line graph, we add two new nodes and their associated edges, which will be a total of $O(|V|)$. Overall, the new graph has $|E|+2$ nodes and $O(|E|^2+|V|)$ edges. Finally, to identify $\hat{p}^*$, we only need to keep track of which nodes are associated with edges in p^* in the original graph, which requires $O(|V|)$

additional time to convert the node labels. This means that f can be computed in polynomial time. To compute g, we only need to maintain a mapping of the edges in G to the nodes in $\hat{G}$, which will take $O(|E|)$ time to populate. Converting the solution to Optimal Force Path Remove to a solution to Optimal Force Path Cut will take $O(|V'|) = O(|E|)$ lookups in the mapping. Thus, both f and g can be computed in polynomial time.

By Lemma C.2, if V' is the optimal solution for $f(\mathcal{A})$, then $g(V')$ is the optimal solution for $\mathcal{A}$, which means condition (2) is satisfied with $\alpha = 1$. In addition, since the solutions of the two problems are the same size—each node removed to solve $f(\mathcal{A})$ corresponds with an edge removed to solve $\mathcal{A}$—we have, for any proposed solution x to $f(\mathcal{A})$,

$$|\text{cost}(x) - \text{opt}(f(\mathcal{A}))| = |\text{cost}(g(x)) - \text{opt}(\mathcal{A})|, \tag{38}$$

satisfying condition (3) with $\beta = 1$. Thus, all conditions of a linear reduction are satisfied. Since Optimal Force Path Cut is APX-hard for unweighted, undirected graphs (Lemma A.1), this implies that Optimal Force Path Remove is APX-hard as well. □

We can trivially reduce from the undirected case to the directed case by creating a directed edge set that includes edges in both directions for each undirected edge in the original graph. Solving Optimal Force Path Remove on the resulting directed graph yields the same solution as solving the Optimal Force Path Remove on the original graph; thus, corresponding solutions in the directed and undirected cases are always equally costly. The formal proof of the following lemma is straightforward, and we omit it for brevity.

Lemma C.4. *Optimal Force Path Remove is APX-hard for directed graphs, including the case where all costs are equal.*

The following theorem is a direct consequence of Lemmas C.3 and C.4.

Theorem C.5. *Optimal Force Path Remove is APX-hard, including the case where all costs are equal.*

C.2 Experiments

We use the same experimental setup as in Section 6.3, in this case only considering the 200th shortest path between the source and target. The baseline method is analogous to the edge removal case: the lowest-cost node along the shortest path is removed until p^* is the shortest path. In all cases, the cost of edge removal is the edge weight, while the cost of node removal is degree. For node removal, we only consider values of p^* that have viable solutions (i.e., p^* is the shortest path from s to t in the subgraph induced on the nodes used by p^*).

Figure 10 shows results on unweighted graphs. We consider the equal-weighted case, without random weights added, as this case highlights the greatest difference between edge cuts and node removal. We did not consider cliques, since there is never a viable solution when p^* is more than one hop. Results for edge removal are shown for comparison. There are a few substantial differences between relative performance using edge removal and node removal. For example, PATHATTACK provides a much more significant cost reduction with node removal rather than edge removal for Erdős–Rényi graphs. In the vast majority of these cases (99/100 for edge removal, 97/100 for node removal), randomized PATHATTACK finds the optimal solution. This suggests that targeting the low-degree nodes—as the baseline does for node removal—is not an effective strategy in this context; i.e., the increased cost of removing higher-degree nodes pays off by removing more competing paths. Another major difference is that the baseline outperforms PATHATTACK for lattices with equal weights. In this case, while PATHATTACK finds the optimal solution with edge removal in 94% of cases, it is only optimal 20% of the time with node removal. Since degrees are

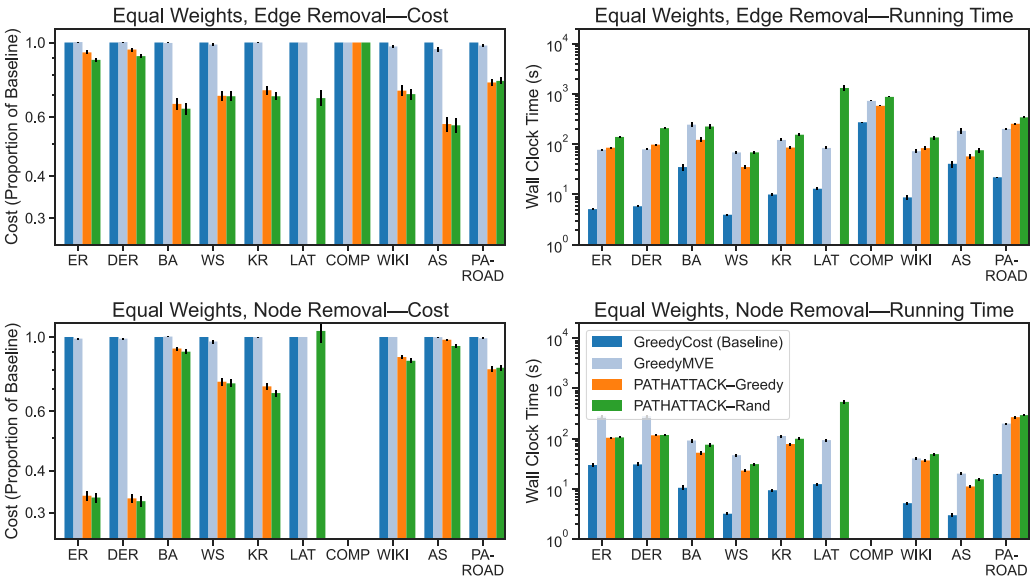

Fig. 10. Results on unweighted networks when PATHATTACK removes nodes rather than edges. Results are shown where all edges have equal weight and removal cost, in terms of attack cost (left column) and running time (right column). Lower is better for both metrics. Bar heights are means across these trials and error bars are standard errors. Results are shown for edge removal (top row) as a comparison to the results for node removal (bottom row). For (D)ER graphs, PATHATTACK yields much more substantial improvement with node removal than edge removal, while for lattices PATHATTACK actually underperforms the baseline. This demonstrates the variation in effectiveness of the baseline (greedily removing low-degree nodes) over differences in topology.

equal, GreedyCost simply removes the first node that deviates from p^*, which works well for unweighted lattices, while the analogous strategy for edge removal is suboptimal. We see a similar phenomenon with the autonomous system graph: the baseline method of removing low-degree nodes yields a near-optimal solution, suggesting that focusing on disrupting the intermediate low-degree nodes between the endpoints and the hubs is a cost-effective strategy. While attacking high-degree nodes is useful for disconnecting networks with skewed degree distributions [2], the degree-based cost and focus on making a particular path shortest make a different strategy ideal here.

Figure 11 illustrates results on real weighted graphs. Unlike the autonomous system graph, here the power grid and road networks outperform the baseline more significantly using node removal. Here, the diversity of edge weights among the real graphs—and their contribution to path length—makes greedily selecting low-cost edges a more effective heuristic than selecting low-degree nodes.

D DATASET FEATURES

Our experiments were run on several synthetic and real networks across different edge-weight distributions. All networks are undirected except DER, WIKI, and LBL. We described the edge-weight distributions in Section 6 of the article. Table 1 provides summary statistics of the synthetic networks. Modularity is computed after performing community detection with the Louvain method [7]. Betweenness centrality is estimated based on the shortest paths between 100 randomly selected source-destination pairs.

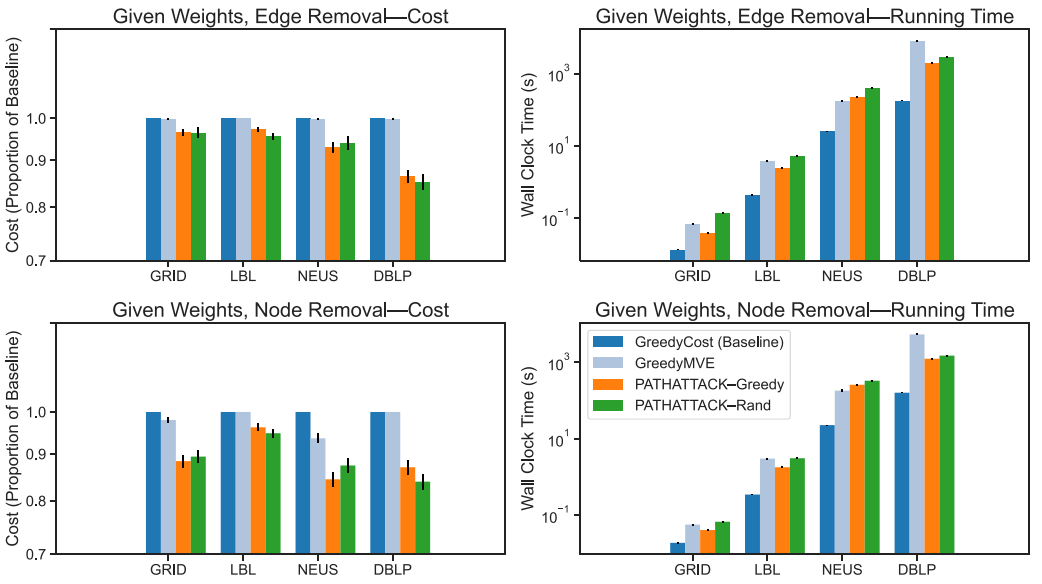

Fig. 11. Results on weighted networks when PATHATTACK removes nodes rather than edges. Results are shown in terms of attack cost (left column) and running time (right column). Lower is better for both metrics. Bar heights are means across these trials and error bars are standard errors. Results are shown for edge removal (top row) as a comparison to the results for node removal (bottom row). Unlike the unweighted graphs, PATHATTACK outperforms the baseline by a higher margin with node removal than with edge removal.

We ran experiments on both weighted and unweighted real networks. In cases of unweighted networks, we added edge weights from the same distributions as the synthetic networks. Below is a brief description of each network used. Table 2 summarizes the properties of each network.

The unweighted networks are:

- Wikispeedia graph (WIKI): The network consists of web pages (nodes) and connections (edges) created from the user-generated paths in the Wikispeedia game [53]. Available at https://snap.stanford.edu/data/wikispeedia.html.
- Oregon autonomous system network (AS): Nodes represent autonomous systems of routers and edges denote communication between the systems [34]. The dataset was collected at the University of Oregon on March 31, 2001. Available at https://snap.stanford.edu/data/Oregon-1.html.
- Pennsylvania road network (PA-ROAD): Nodes are intersections in Pennsylvania, connected by edges representing roads [35]. Available at https://snap.stanford.edu/data/roadNet-PA.html.

The weighted networks are:

- Central Chilean Power Grid (GRID): Nodes represent power plants, substations, taps, and junctions in the Chilean power grid. Edges represent transmission lines, with distances in kilometers [32]. The capacity of each line in kilovolts is also provided. Available at https://figshare.com/collections/An_in-depth_network_structural_data_and_hourly_activity_on_the_Central_Chilean_power_grid/4053374.
- Lawrence Berkeley National Laboratory network data (LBL): A graph of computer network traffic, which includes counts of the number of connections between machines over

Table 1. Properties of the Synthetic Networks Used in Our Experiments

Network	Nodes	Edges	$\langle k \rangle$	σ_k	κ	τ	$\triangle$
ER	16,000 ±0.0	159,840 ±374.025	19.98 ±0.047	4.462 ±0.017	0.001 ±0.0	0.001 ±0.0	1,318 ±42.488
DER	16,000 ±0.0	319,700 ±575.92	39.963 ±0.072	6.32 ±0.031	0.001 ±0.0	0.001 ±0.0	10,667.2 ±137.267
BA	16,000 ±0.0	159,900 ±0.0	19.988 ±0.0	24.476 ±0.289	0.008 ±0.0	0.007 ±0.0	17,089.1 ±436.128
WS	16,000 ±0.0	160,000 ±0.0	20 ±0.0	0.627 ±0.005	0.67 ±0.001	0.668 ±0.001	677,850.8 ±820.131
KR	16,345.1 ±17.038	159,645 ±80.722	19.534 ±0.015	16.063 ±1.195	0.003 ±0.0	0.004 ±0.001	7,605.6 ±1,975.135
LAT	81,225 ±0.0	161,880 ±0.0	3.986 ±0.0	0.118 ±0.0	0 ±0.0	0 ±0.0	0 ±0.0
COMP	565 ±0.0	159,330 ±0.0	564 ±0.0	0.0 ±0.0	1.0 ±0.0	1.0 ±0.0	29,900,930 ±0.0

Network	d	BC	Q	$\langle \ell \rangle$	σ_ℓ
ER	5 ±0.0	1e−4 ±1.5e−7	0.182 ±0.001	4.556 ±0.057	0.561 ±0.029
DER	5.0 ±0.0	1e−4 ±1.5e−7	0.053 ±0.036	4.605 ±0.038	0.53 ±0.026
BA	4.3 ±0.458	1e−4 ±4e−7	0.198 ±0.001	4.19 ±0.054	0.502 ±0.038
WS	9.6 ±0.49	3e−4 ±2e−6	0.944 ±0.0	7.445 ±0.084	1.152 ±0.098
KR	7 ±0.632	1e−4 ±0.0	0.193 ±0.006	4.533 ±0.046	0.625 ±0.022
LAT	568 ±0.0	0.0023 ±1e−6	0.951 ±0.0	182.41 ±0.0	85.702 ±0.0
COMP	1 ±0.0	0 ±0	0 ±0.0	1 ±0.0	0 ±0.0

For each random graph model, we generate 100 networks. Note that the number of edges across the different networks is ≈ 160K. The table shows the average degree ($\langle k \rangle$), standard deviation of the degree (σ_k), average clustering coefficient (κ), transitivity (τ), number of triangles ($\triangle$), diameter (d), average betweenness centrality (BC), modularity (Q), average shortest path distance($\langle \ell \rangle$) and standard deviation of shortest path distance (σ_ℓ). The ± values show the standard deviation across 100 runs of each random graph model.

time. Counts are inverted for use as distances. Available at https://www.icir.org/enterprise-tracing/download.html.

- Northeast US Road Network (NEUS): Nodes are intersections in the northeastern part of the United States, interconnected by roads (edges), with weights corresponding to distance in kilometers. Available at https://www.diag.uniroma1.it/~challenge9/download.shtml.
- DBLP coauthorship graph (DBLP): This is a coauthorship network [5]. We invert the number of coauthored papers to create a distance (rather than similarity) between the associated authors. Available at https://www.cs.cornell.edu/~arb/data/coauth-DBLP/.

Table 2. Properties of the Real Networks Used in Our Experiments

Network	Nodes	Edges	$\langle k \rangle$	σ_k	κ	τ	$\triangle$
GRID	347	444	2.559	1.967	0.086	0.087	40
LBL	3, 150	9, 459	6.005	25.654	0.100	0.005	1,821
WIKI	4, 589	106, 644	46.478	69.892	0.274	0.102	550, 544
AS	10, 670	22, 002	4.124	31.986	0.297	0.009	17, 144
PA-ROAD	1, 087, 562	1, 541, 514	2.834	1.016	0.046	0.059	67, 112
NEUS	1, 524, 453	1, 934, 010	2.537	0.950	0.022	0.030	37, 012
DBLP	1, 659, 954	8, 119, 276	9.782	22.359	0.643	0.165	26, 781, 590

Network	d	BC	Q	$\langle \ell \rangle$	σ_ℓ
GRID	23	0.020	0.832	9.09	3.203
LBL	11	8e−4	0.729	4.68	0.936
WIKI	5	3e−4	0.371	3.51	0.499
AS	10	2e−4	0.627	4.59	0.918
PA-ROAD	794	2e−4	0.989	293.24	155.280
NEUS	2098	4e−4	0.992	758.04	404.963
DBLP	22	2.25e−6	0.754	6.73	1.156

For each network, we are listing the average degree ($\langle k \rangle$), standard deviation of the degree (σ_k), average clustering coefficient (κ), transitivity (τ), number of triangles ($\triangle$), diameter (d), betweenness centrality (BC), modularity (Q), average shortest path distance($\langle \ell \rangle$) and standard deviation of shortest path distance (σ_ℓ).

REFERENCES

[1] Ravindra K. Ahuja and James B. Orlin. 2001. Inverse optimization. *Operations Research* 49, 5 (2001), 771–783.

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 2000. Error and attack tolerance of complex networks. *Nature* 406, 6794 (2000), 378–382.

[3] Yash P. Aneja and Kunhiraman P. K. Nair. 1978. The constrained shortest path problem. *Naval Research Logistics Quarterly* 25, 3 (1978), 549–555.

[4] Walid Ben-Ameur and José Neto. 2006. A constraint generation algorithm for large scale linear programs using multiple-points separation. *Mathematical Programming* 107, 3 (2006), 517–537.

[5] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.

[6] Dimitris Bertsimas, Ebrahim Nasrabadi, and James B. Orlin. 2016. On the power of randomization in network interdiction. *Operations Research Letters* 44, 1 (2016), 114–120.

[7] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.

[8] Aleksandar Bojchevski and Stephan Günnemann. 2019. Adversarial attacks on node embeddings via graph poisoning. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*. PMLR, 695–704.

[9] Martim Brandão, Amanda Coles, and Daniele Magazzeni. 2021. Explaining path plan optimality: Fast explanation methods for navigation meshes using full and incremental inverse optimization. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS'21)*, Vol. 31. AAAI Press, Palo Alto, CA, 56–64.

[10] Mikael Call and Kaj Holmberg. 2011. Complexity of inverse shortest path routing. In *Network Optimization (INOC'11)*. Springer, Berlin, 339–353.

[11] Heng Chang, Yu Rong, Tingyang Xu, Wenbing Huang, Honglei Zhang, Peng Cui, Wenwu Zhu, and Junzhou Huang. 2020. A restricted black-box adversarial framework towards attacking graph embedding models. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'20)*. AAAI Press, Palo Alto, CA, 3389–3396.

[12] Jinyin Chen, Lihong Chen, Yixian Chen, Minghao Zhao, Shanqing Yu, Qi Xuan, and Xiaoniu Yang. 2019. GA-based Q-attack on community detection. *IEEE Transactions on Computational Social Systems* 6, 3 (2019), 491–503.

[13] Jinyin Chen, Yixian Chen, Lihong Chen, Minghao Zhao, and Qi Xuan. 2020. Multiscale evolutionary perturbation attack on community detection. *IEEE Transactions on Computational Social Systems* 8, 1 (2020), 62–75.

[14] Jinyin Chen, Yangyang Wu, Xuanheng Xu, Yixian Chen, Haibin Zheng, and Qi Xuan. 2018. Fast Gradient Attack on Network Embedding. https://doi.org/10.48550/ARXIV.1809.02797
[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. Hash tables. In *Introduction to Algorithms*. MIT Press, Cambridge, MA, 219–243.
[16] Tingting Cui and Dorit S. Hochbaum. 2010. Complexity of some inverse shortest path lengths problems. *Networks* 56, 1 (2010), 20–29.
[17] William H. Cunningham and Lawrence Tang. 1999. Optimal 3-terminal cuts and linear programming. In *International Conference on Integer Programming and Combinatorial Optimization*. Springer, Springer, Berlin, 114–125.
[18] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis. 1994. The complexity of multiterminal cuts. *SIAM Journal on Computing* 23, 4 (1994), 864–894.
[19] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. 2018. Adversarial attack on graph structured data. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*. PMLR, 1115–1124.
[20] Adrian Deaconu and Laura Ciupala. 2020. Inverse minimum cut problem with lower and upper bounds. *Mathematics* 8, 9 (2020), 10 pages.
[21] Bistra Dilkina, Katherine J. Lai, and Carla P. Gomes. 2011. Upgrading shortest paths in networks. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'11)*. Springer, Berlin, 76–91.
[22] David Eppstein. 1998. Finding the k shortest paths. *SIAM Journal on Computing* 28, 2 (1998), 652–673.
[23] Valeria Fionda and Giuseppe Pirro. 2017. Community deception or: How to stop fearing community detection algorithms. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2017), 660–673.
[24] Peter Gács and Laszlo Lovász. 1981. Khachiyan's algorithm for linear programming. In *Mathematical Programming at Oberwolfach*. Springer, Berlin, 61–68.
[25] Martin Grötschel, László Lovász, and Alexander Schrijver. 1981. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica* 1, 2 (1981), 169–197.
[26] Manish Jain, Dmytro Korzhyk, Ondřej Vaněk, Vincent Conitzer, Michal Pěchouček, and Milind Tambe. 2011. A double oracle algorithm for zero-sum security games on graphs. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'11)*. IFAAMAS, Richland, SC, 327–334.
[27] Yiwei Jiang, Longcheng Liu, Biao Wu, and Enyu Yao. 2010. Inverse minimum cost flow problems under the weighted Hamming distance. *European Journal of Operational Research* 207, 1 (2010), 50–54.
[28] Zhongyuan Jiang, Jing Li, Jianfeng Ma, and Philip S. Yu. 2020. Similarity-based and Sybil attack defended community detection for social networks. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 12 (2020), 3487–3491.
[29] Myungsoo Jun and Raffaello D'Andrea. 2003. Path planning for unmanned aerial vehicles in uncertain and adversarial environments. In *Cooperative Control: Models, Applications and Algorithms*. Springer US, Boston, MA, 95–110.
[30] W. Philip Kegelmeyer, Jeremy D. Wendt, and Ali Pinar. 2018. *An Example of Counter-adversarial Community Detection Analysis*. Technical Report SAND2018-12068. Sandia National Laboratories. https://doi.org/10.2172/1481570
[31] Subhash Khot and Oded Regev. 2008. Vertex cover might be hard to approximate to within 2- ε. *Journal of Computer and System Sciences* 74, 3 (2008), 335–349.
[32] Heetae Kim, David Olave-Rojas, Eduardo Álvarez-Miranda, and Seung-Woo Son. 2018. In-depth data on the network structure and hourly activity of the Central Chilean power grid. *Scientific Data* 5, 1 (2018), 1–10.
[33] Yin Tat Lee and Aaron Sidford. 2015. Efficient inverse maintenance and faster algorithms for linear programming. In *IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS'15)*. IEEE, Piscataway, NJ, 230–249.
[34] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05)*. ACM, New York, NY, 177–187.
[35] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
[36] Joshua Letchford and Yevgeniy Vorobeychik. 2013. Optimal interdiction of attack plans. In *Proceedings of the 12th International Conference on Autonomous Agents and Multi-Agent Systems*. IFAAMAS, Richland, SC, 199–206.
[37] Jia Li, Honglei Zhang, Zhichao Han, Yu Rong, Hong Cheng, and Junzhou Huang. 2020. Adversarial attack on community detection by hiding individuals. In *Proceedings of The Web Conference (WWW'20)*. ACM, New York, NY, 917–927.
[38] Longcheng Liu and Jianzhong Zhang. 2006. Inverse maximum flow problems under the weighted Hamming distance. *Journal of Combinatorial Optimization* 12, 4 (2006), 395–408.
[39] Sourav Medya, Petko Bogdanov, and Ambuj Singh. 2016. Towards scalable network delay minimization. In *IEEE 16th International Conference on Data Mining (ICDM'16)*. IEEE, Piscataway, NJ, 1083–1088.

[40] Sourav Medya, Sayan Ranu, Jithin Vachery, and Ambuj Singh. 2018. Noticeable network delay minimization via node upgrades. *Proceedings of the VLDB Endowment* 11, 9 (2018), 988–1001.
[41] Adam Meyerson and Brian Tagiku. 2009. Minimizing average shortest path distances via shortcut edge addition. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX–RANDOM'09)*. Springer, Berlin, 272–285.
[42] Benjamin A. Miller, Zohair Shafi, Wheeler Ruml, Yevgeniy Vorobeychik, Tina Eliassi-Rad, and Scott Alfeld. 2021. PATHATTACK: Attacking shortest paths in complex networks. In *Machine Learning and Knowledge Discovery in Databases. Research Track (ECML PKDD'21)*, Nuria Oliver, Fernando Pérez-Cruz, Stefan Kramer, Jesse Read, and Jose A. Lozano (Eds.). Springer International Publishing, Cham, 532–547.
[43] Shishir Nagaraja. 2010. The impact of unlinkability on adversarial community detection: Effects and countermeasures. In *Privacy Enhancing Technologies (PETS'10)*. Springer, Berlin, 253–272.
[44] Enrico Nardelli, Guido Proietti, and Peter Widmayer. 2001. A faster computation of the most vital edge of a shortest path. *Information Processing Letters* 79, 2 (2001), 81–85.
[45] Enrico Nardelli, Guido Proietti, and Peter Widmayer. 2003. Finding the most vital node of a shortest path. *Theoretical Computer Science* 296, 1 (2003), 167–177.
[46] Ran Raz and Shmuel Safra. 1997. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC'97)*. ACM, New York, NY, 475–484.
[47] Javad Tayyebi and Massoud Aman. 2016. Efficient algorithms for the reverse shortest path problem on trees under the Hamming distance. *Yugoslav Journal of Operations Research* 27, 1 (2016), 46–60.
[48] Hanghang Tong, B. Aditya Prakash, Tina Eliassi-Rad, Michalis Faloutsos, and Christos Faloutsos. 2012. Gelling, and melting, large graphs by edge manipulation. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12)*. ACM, New York, NY, 245–254.
[49] Vijay V. Vazirani. 2003. *Approximation Algorithms*. Springer, Berlin.
[50] Vijay V. Vazirani. 2003. Rounding applied to set cover. In *Approximation Algorithms*. Springer, Berlin, 118–123.
[51] Marcin Waniek, Tomasz P. Michalak, Michael J. Wooldridge, and Talal Rahwan. 2018. Hiding individuals and communities in a social network. *Nature Human Behaviour* 2, 2 (2018), 139–147.
[52] Alan Washburn and Kevin Wood. 1995. Two-person zero-sum games for network interdiction. *Operations Research* 43, 2 (1995), 243–251.
[53] Robert West, Joelle Pineau, and Doina Precup. 2009. Wikispeedia: An online game for inferring semantic distances between concepts. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*. AAAI Press, Palo Alto, CA, 1598–1603.
[54] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. 2019. Adversarial examples for graph data: Deep insights into attack and defense. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. IJCAI, 4816–4823.
[55] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. 2019. Topology attack and defense for graph neural networks: An optimization perspective. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. IJCAI, 3961–3967.
[56] Shaoji Xu and Jianzhong Zhang. 1995. An inverse problem of the weighted shortest path problem. *Japan Journal of Industrial and Applied Mathematics* 12, 1 (1995), 47–59.
[57] Bin-wu Zhang and Qin Wang. 2008. Inverse shortest path problems under Hamming distance. *Journal of Hohai University (Natural Sciences)* 36, 4 (July 2008), 571–574.
[58] Jianzhong Zhang and Yixun Lin. 2003. Computation of the reverse shortest-path problem. *Journal of Global Optimization* 25, 3 (2003), 243–261.
[59] Jianzhong Zhang, Zhongfan Ma, and Chao Yang. 1995. A column generation method for inverse shortest path problems. *Zeitschrift für Operations Research* 41, 3 (1995), 347–358.
[60] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*. ACM, New York, NY, 2847–2856.

Received 31 January 2023; revised 30 June 2023; accepted 17 August 2023