PATHATTACK: Attacking Shortest Paths in Complex Networks

Benjamin A. Miller[✉][1[0000−0002−1649−1401]], Zohair Shafi[1[0000−0001−6154−1466]], Wheeler Ruml[2[0000−0002−1308−2311]], Yevgeniy Vorobeychik[3[0000−0003−2471−5345]], Tina Eliassi-Rad[1[0000−0002−1892−1188]], and Scott Alfeld[4[0000−0001−8446−4993]]

[1] Northeastern University, Boston MA 02115, USA
{miller.be, shafi.z, t.eliassirad}@northeastern.edu
[2] University of New Hampshire, Durham NH 03824, USA
ruml@cs.unh.edu
[3] Washington University in St. Louis, St. Louis MO 63130, USA
yvorobeychik@wustl.edu
[4] Amherst College, Amherst MA 01002, USA
salfeld@amherst.edu

Abstract. Shortest paths in complex networks play key roles in many applications. Examples include routing packets in a computer network, routing traffic on a transportation network, and inferring semantic distances between concepts on the World Wide Web. An adversary with the capability to perturb the graph might make the shortest path between two nodes route traffic through advantageous portions of the graph (e.g., a toll road he owns). In this paper, we introduce the Force Path Cut problem, in which there is a specific route the adversary wants to promote by removing a low-cost set of edges in the graph. We show that Force Path Cut is NP-complete. It can be recast as an instance of the Weighted Set Cover problem, enabling the use of approximation algorithms. The size of the universe for the set cover problem is potentially factorial in the number of nodes. To overcome this hurdle, we propose the `PATHATTACK` algorithm, which via constraint generation considers only a small subset of paths—at most 5% of the number of edges in 99% of our experiments. Across a diverse set of synthetic and real networks, the linear programming formulation of Weighted Set Cover yields the optimal solution in over 98% of cases. We also demonstrate running time vs. cost tradeoff using two approximation algorithms and greedy baseline methods. This work expands the area of adversarial graph mining beyond recent work on node classification and embedding.

Keywords: Adversarial graph perturbation · Shortest path · Constraint generation.

1 Introduction

In a variety of applications, finding shortest paths among interconnected entities is an important task. Whether routing traffic on a road network, packets in a

computer network, ships in a maritime network, or identifying the "degrees of separation" between two actors, locating the shortest path is often key to making efficient use of the interconnected entities. By manipulating the shortest path between two popular entities—e.g., people or locations—those along the altered path could have much to gain from the increased exposure. Countering such behavior is important, and understanding vulnerability to such manipulation is a step toward more robust graph mining.

In this paper, 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 go through a preferred path. The adversary has a fixed budget and achieves this goal by cutting edges, each of which has a cost for removal. We show that this problem is NP-complete via a reduction from the 3-Terminal Cut problem [5]. To solve 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 paths to target for removal. While these algorithms only guarantee an approximately optimal solution in general, `PATHATTACK` yields the lowest-cost solution in a large majority of our experiments.

The main contributions of the paper are as follows: (1) We formally define Force Path Cut and show that it is NP complete. (2) We demonstrate that approximation algorithms for Weighted Set Cover can be leveraged to solve the Force Path Cut problem. (3) We identify an oracle to judiciously select paths to consider for removal, avoiding the combinatorial explosion inherent in naïvely enumerating all paths. (4) We propose the `PATHATTACK` algorithm, which integrates these elements into an attack strategy. (5) We summarize the results of over 20,000 experiments on synthetic and real networks, in which `PATHATTACK` identifies the optimal attack in over 98% of the time.

2 Problem Statement

We are given a graph $G = (V, E)$, where the vertex set V is a set of N entities and E is a set of M undirected edges representing the ability to move between the entities. In addition, we have nonnegative edge weights $w : E \to \mathbb{R}_{\geq 0}$ denoting the expense of traversing edges (e.g., distance or time).

We are also given two nodes $s, t \in V$. An adversary has the goal of routing traffic from s to t along a given path p^*. This adversary removes edges with full knowledge of G and w, and each edge has a cost $c : E \to \mathbb{R}_{\geq 0}$ of being removed. Given a budget b, the adversary's objective is to remove a set of edges $E' \subset E$ such that $\sum_{e \in E'} c(e) \leq b$ and p^* is the exclusive shortest path from s to t in the resulting graph $G' = (V, E \setminus E')$. We refer to this problem as *Force Path Cut*.

We show that this problem is computationally intractable in general by reducing from the 3-Terminal Cut problem, which is known to be NP-complete [5]. In 3-Terminal Cut, we are given a graph $G = (V, E)$ with weights w, a budget $b \geq 0$, and three terminal nodes $s_1, s_2, s_3 \in V$, and are asked whether a set of

edges can be removed such that (1) the sum of the weights of the removed edges is at most b and (2) s_1, s_2, and s_3 are disconnected in the resulting graph (i.e., there is no path connecting any two terminals). Given that 3-Terminal Cut is NP-complete, we prove the following theorem.

Theorem 1. *Force Path Cut is NP-complete for undirected graphs.*

Here we provide an intuitive sketch of the proof; the formal proof is included in the supplementary material.

Proof Sketch. Suppose we want to solve 3-Terminal Cut for a graph $G = (V, E)$ with weights w, where the goal is to find $E' \subset E$ such that the terminals are disconnected in $G' = (V, E \setminus E')$ and $\sum_{e \in E'} w(e) \leq b$. We first consider the terminal nodes: If any pair of terminals shares an edge, that edge must be included in E' regardless of its weight; the terminals would not be disconnected if this edge remains. Note also that for 3-Terminal Cut, edge weights are edge removal costs; there is no consideration of weights as distances. If we add new edges between the terminals that are costly to both traverse and remove, then forcing one of these new edges to be the shortest path requires removing any other paths between the terminal nodes. This causes the nodes to be disconnected in the original graph. We will use a large weight for this purpose: $w_{\text{all}} = \sum_{e \in E} w(e)$, the sum of all weights in the original graph.

We reduce 3-Terminal Cut to Force Path Cut as follows. Create a new graph $\hat{G} = (V, \hat{E})$, where $\hat{E} = E \cup \{\{s_1, s_2\}, \{s_1, s_3\}, \{s_2, s_3\}\}$—i.e., $\hat{G}$ is the input graph with edges between the terminals added if they did not already exist. In addition, create new weights $\hat{w}$ where, for some $\epsilon > 0$, $\hat{w}(\{s_1, s_2\}) = \hat{w}(\{s_2, s_3\}) = w_{\text{all}} + 2\epsilon$ and $\hat{w}(\{s_1, s_3\}) = 2w_{\text{all}} + 3\epsilon$, and $\hat{w}(e) = w(e)$ for all other edges. Let the edge removal costs in the new graph be equal to the weights, i.e., $\hat{c}(e) = \hat{w}(e)$ for all $e \in \hat{E}$. Finally, let the target path consist only of the edge from s_1 to s_3, i.e., $s = s_1$, $t = s_3$, and $p^* = (s, t)$.

If we could solve Force Path Cut on $\hat{G}$ with weights $\hat{w}$ and costs $\hat{c}$, it would yield a solution to 3-Terminal Cut. We can assume the budget b is at most w_{all}, since this would allow the trivial solution of removing all edges and any additional budget would be unnecessary. If any edges exist between terminals in the original graph G, they must be included in the set of edges to remove, and their weights must be removed from the budget, yielding a new budget $\hat{b}$. Using this new budget for Force Path Cut, we will find a solution $\hat{E}' \subset \hat{E}$ if and only if there is a solution $E' \subset E$ for 3-Terminal Cut. A brief explanation of the reasoning is as follows:

- When we solve Force Path Cut, we are forcing an edge with a very large weight to be on the shortest path. If any path from s_1 to s_3 from the original graph remained, it would be shorter than (s_1, s_3). In addition, if any path from G between s_1 and s_2 remained, its length would be at most w_{all}, and thus a path from s_1 to s_3 that included s_2 would have length at most $2w_{\text{all}} + 2\epsilon$. This would mean (s_1, s_3) is not the shortest path between s_1 and s_3. A similar argument holds for paths between s_2 and s_3. Thus, no paths can remain between the terminals if we find a solution for Force Path Cut.

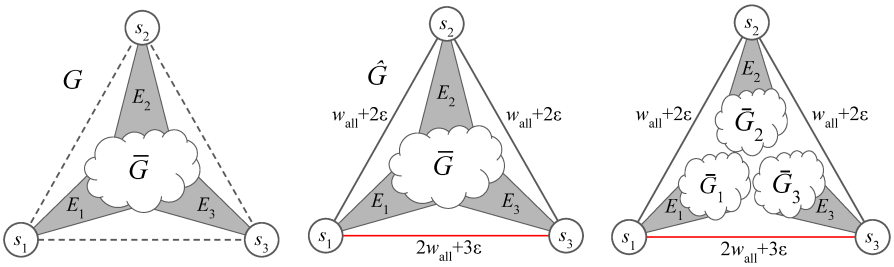

Fig. 1. Conversion from input to 3-Terminal Cut to Force Path Cut. The initial graph (left) includes 3 terminal nodes s_1, s_2, and s_3, which are connected to the rest of the graph by edges E_1, E_2, and E_3, respectively. The dashed lines indicate the possibility of edges between terminals. The input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

– If a solution exists for 3-Terminal Cut in G, it will yield the solution for Force Path Cut in $\hat{G}$. Any edge added to the graph to create $\hat{G}$ would be more costly to remove than removing all edges from the original G, so none will be removed. With all original paths between terminals removed, the only ones remaining from s_1 to s_3 are (s_1, s_3) and (s_1, s_2, s_3), the former of which is shortest, thus yielding a solution to Force Path Cut.

Figure 1 illustrates the aforementioned procedure. Note that the procedure would yield a solution to 3-Terminal Cut even if Force Path Cut allows for ties with p^*, so Force Path Cut is NP-complete in this case as well. □

3 Proposed Method: PATHATTACK

While solving Force Path Cut is computationally intractable, we formulate the problem in a way that enables the use of established approximation algorithms.

3.1 Path Cutting as Set Cover

The success condition of Force Path Cut is that all paths from s to t aside from p^* must be strictly longer than p^*. This is an example of the (Weighted) Set Cover problem. In Weighted Set Cover, we are given a discrete universe $\mathcal{U}$ and a set of subsets of the universe $\mathcal{S}$, $S \subset \mathcal{U}$ for all $S \in \mathcal{S}$, where each set has a cost $c(S)$. The goal is to choose those subsets whose aggregate cost is within a budget yet whose union equals the universe. In Force Path Cut, the elements of the universe to cover are the paths and the sets represent edges: each edge corresponds to a set containing all paths from s to t on which it lies. Including this set in the cover

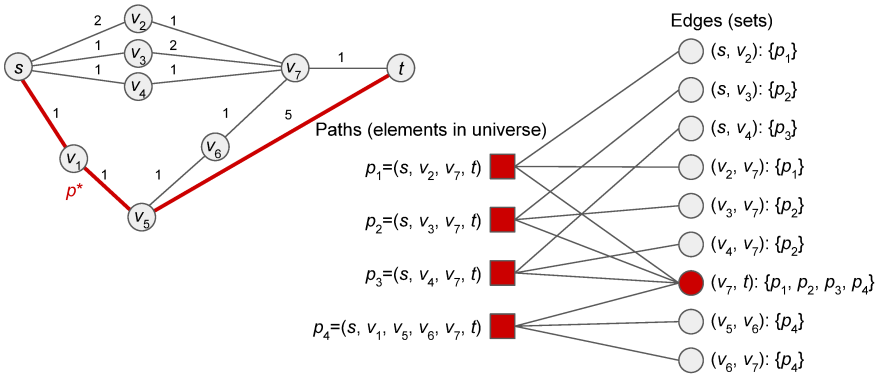

Fig. 2. The Force Path Cut problem is an example of the Weighted Set Cover problem. In the bipartite graph on the right, the square nodes represent paths and the circle nodes represent edges. Note that edges along p^* are not included. When the red-colored circle (i.e., edge (v_7, t)) is removed, then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

implies removing the edge, thus covering the elements (i.e., cutting the paths). Figure 2 shows how Force Path Cut is an example of Weighted Set Cover.

While Set Cover is NP-complete, there are known approximation algorithms to get a solution within a factor of $O(\log |\mathcal{U}|)$ of the optimal cost. The challenge in our case is that the universe may be extremely large. We address this challenge over the remainder of this section.

3.2 Linear Programming Formulation

In this section, we focus on minimizing cost without explicitly considering a budget. In practice, the adversary would run one of the optimization algorithms, compare budget and cost, and decide whether the attack is possible given resource constraints. Let $\mathbf{c} \in \mathbb{R}^M_{\geq 0}$ be a vector of edge costs, where each entry in the vector corresponds to an edge in the graph. We want to minimize the sum of the costs of edges that are cut, which is the dot product of $\mathbf{c}$ with a binary vector indicating which edges are cut, denoted by $\Delta \in \{0,1\}^M$. This means that we optimize over values of Δ under constraints that (1) p^* is not cut and (2) all other paths from s to t not longer than p^* *are* cut. We represent paths in this formulation by binary indicator vectors—i.e., the vector $\mathbf{x}_p \in \{0,1\}^M$ that represents path p is 1 at entries corresponding to edges in p and 0 elsewhere. Since any edge can only occur once, we only consider simple paths—those without cycles—which is sufficient for our purposes. If there is one index that is one in both Δ and $\mathbf{x}_p$, the path p is cut. Let P_p be the set of all paths in G from p's source to its destination that are no longer than p. The integer linear program formulation of Force Path Cut is as follows:

$$\hat{\Delta} = \arg\min_{\Delta} \mathbf{c}^\top \Delta \tag{1}$$

$$\text{s.t. } \Delta \in \{0,1\}^M \tag{2}$$

$$\mathbf{x}_p^\top \Delta \geq 1 \ \forall p \in P_{p^*} \setminus \{p^*\} \tag{3}$$

$$\mathbf{x}_{p^*}^\top \Delta = 0. \tag{4}$$

Constraint (3) ensures that any path not longer than (thus competing with) p^* will be cut, and constraint (4) forbids cutting p^*. As mentioned previously, P_{p^*} may be extremely large, which we address in Section 3.3.

The formulation (1)–(4) is analogous to the formulation of Set Cover as an integer program [19]. The goal is to minimize the cost of covering the universe — i.e., for each element $x \in \mathcal{U}$, at least one set $S \in \mathcal{S}$ where $x \in S$ is included. Letting δ_S be a binary indicator of the inclusion of subset S, the integer program formulation of Set Cover is

$$\hat{\hat{\delta}} = \arg\min_{\delta} \sum_{S \in \mathcal{S}} c(S)\delta_S \tag{5}$$

$$\text{s.t. } \delta_S \in \{0,1\} \ \forall S \in \mathcal{S} \tag{6}$$

$$\sum_{S \in \{S' \in \mathcal{S} | x \in S\}} \delta_S \geq 1 \ \forall x \in \mathcal{U}. \tag{7}$$

Equations (1), (2), and (3) are analogous to (5), (6), and (7), respectively. The constraint (4) can be incorporated by not allowing some edges to be cut, which manifests itself as removing some subsets from $\mathcal{S}$.

With Force Path Cut formulated as Set Cover, we consider two approximation algorithms. The first method, `GreedyPathCover`, iteratively adds the most cost-effective subset: that with the largest number of uncovered elements per cost. In Force Path Cut, this is equivalent to iteratively cutting the edge that removes the most paths per cost. The pseudocode is shown in Algorithm 1. We have a fixed set of paths $P \subset P_{p^*} \setminus \{p^*\}$. 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 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 in the worst case is $O(|P|N)$. Finding the most cost-effective edge takes $O(M)$ time with a naïve implementation, and this portion is run at most once per path, leading to an overall running time of $O(|P|(N+M))$. 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 $O(\log M)$ time, for an overall running time of $O(|P|N \log M)$. The worst-case approximation factor is the harmonic function of the size of the universe [19],

i.e., $H_{|\mathcal{U}|} = \sum_{n=1}^{|\mathcal{U}|} 1/n$, which implies that the `GreedyPathCover` algorithm has a worst-case approximation factor of $H_{|P|}$. As we discuss in Section 3.4, this approximation factor extends to the overall Force Path Cut problem.

Input: Graph $G = (V, E)$, costs c, target path p^*, path set P
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 *edges* e *in* p *and not* p^* **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'

Algorithm 1: GreedyPathCover

The second approximation algorithm we consider involves relaxing the integer constraint into the reals and rounding the resulting solution. We refer to this algorithm as `LP-PathCover`. In this case, we replace (2) with the condition $\Delta \in [0, 1]^M$ and get a $\hat{\Delta}$ that may contain non-integer entries. Following the procedure in [19], we apply randomized rounding as follows for each edge e:

1. Treat the corresponding entry $\hat{\Delta}_e$ as a probability.
2. Draw $\lceil \ln(4|P|) \rceil$ independent Bernoulli random variables w/ probability $\hat{\Delta}_e$.
3. Cut e if and only if at least one random variable from step 2 is 1.

If the result either does not cut all paths or is too large—i.e., greater than $4 \ln(4|P|)$ times the fractional (relaxed) cost—the procedure is repeated. These

conditions are both satisfied with probability greater than 1/2, so the expected number of attempts to get a valid solution is less than 2. By construction, the approximation factor is $4 \ln (4|P|)$ in the worst case. The running time is dominated by running the linear program; the remainder of the algorithm is (with high probability) linear in the number of edges and logarithmic in the number of constraints $|P|$. Algorithm 2 provides the pseudocode for `LP-PathCover`.

Input: Graph $G = (V, E)$, costs $\mathbf{c}$, path p^*, path set P
Output: Binary vector Δ denoting edges to cut
$\hat{\Delta} \leftarrow$ relaxed cut solution to (1)–(3) with paths P;
$\Delta \leftarrow \mathbf{0}$;
$E' \leftarrow \emptyset$;
not_cut$\leftarrow$ **True**;
while $\mathbf{c}^\top \Delta > \mathbf{c}^\top \hat{\Delta}(4 \ln (4|P|))$ **or** not_cut **do**
  $E' \leftarrow \emptyset$;
  for $i \leftarrow 1$ *to* $\lceil \ln (4|P|) \rceil$ **do**
    `// randomly select edges based on` $\hat{\Delta}$
    $E_1 \leftarrow \{e \in E$ with probability $\hat{\Delta}_e\}$;
    $E' \leftarrow E' \cup E_1$;
  end
  $\Delta \leftarrow$ indicator vector for E';
  not_cut$\leftarrow$ ($\exists p \in P$ where p has no edge in E');
end
return Δ

Algorithm 2: LP-PathCover

3.3 Constraint Generation

In general, it is intractable to include every path from s to t. Take the example of an N-vertex clique (a.k.a. complete graph) in which all edges have weight 1 except the edge from s to t, which has weight N, and let $p^* = (s, t)$. Since all simple paths other than p^* are shorter than N, all of those paths will be included as constraints in (3), including $(N-2)!$ paths of length $N-1$. If we only explicitly include constraints corresponding to the two- and three-hop paths (a total of $(N-2)^2 + (N-2)$ paths), then the optimal solution will be the same as if we had included all constraints: cut the $N-2$ edges around either s or t that do not directly link s and t. Optimizing using only necessary constraints is the other technique we use to make an approximation of Force Path Cut tractable.

Constraint generation is a technique for automatically building a relatively small set of constraints when the total number is extremely large or infinite [2, 12]. The method requires an oracle that, given a proposed solution, returns a constraint that is being violated. This constraint is then explicitly incorporated into the optimization, which is run again and a new solution is proposed. This

procedure is repeated until the optimization returns a feasible point or determines there is no feasible region.

Given a proposed solution to Force Path Cut—obtained by either approximation algorithm from Section 3.2—we have an oracle to identify unsatisfied constraints in polynomial time. We find the shortest path p in $G' = (V, E \setminus E')$ aside from p^*. If p is not longer than p^*, then cutting p is added as a constraint. We combine this constraint generation oracle with the approximation algorithms to create our proposed method `PATHATTACK`.

3.4 PATHATTACK

Combining the above techniques, we propose the `PATHATTACK` algorithm, which enables flexible computation of attacks to manipulate shortest paths. Starting with an empty set of path constraints, `PATHATTACK` alternates between finding edges to cut and determining whether removal of these edges results in p^* being the shortest path from s to t. Algorithm 3 provides `PATHATTACK`'s pseudocode. Depending on time or budget considerations, an adversary can vary the underlying approximation algorithm.

Input: Graph $G = (V, E)$, cost function c, weights w, target path p^*, flag l
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**
  $P \leftarrow P \cup \{p\}$;
  if l **then**
    $\Delta \leftarrow$ `LP-PathCover`$(G, \mathbf{c}, p^*, P)$;
    $E' \leftarrow$ edges from Δ;
  end
  else
    $E' \leftarrow$ `GreedyPathCover`(G, c, p^*, P);
  end
  $G' \leftarrow (V, E \setminus E')$;
  $p \leftarrow$ shortest path from s to t in G' (not including p^*) using weights w;
end
return E'

Algorithm 3: PATHATTACK

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 optimal solution could potentially be lower-cost than when using all constraints. By the final iteration of PATHATTACK, however, we have a solution to Force Path Cut that is within $H_{|P|}$ of the optimum of the less constrained problem, using $|P|$ from the final iteration. This yields the following proposition:

Proposition 2. *The approximation factor of PATHATTACK-Greedy is at most $H_{|P|}$ times the optimal solution to Force Path Cut.*

A similar argument holds for PATHATTACK-LP, applying the results of [19]:

Proposition 3. *PATHATTACK-LP yields a worst-case $O(\log |P|)$ approximation to Force Path Cut with high probability.*

4 Experiments

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

4.1 Baseline Methods

We consider two simple greedy methods as baselines for assessing performance. 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 where we cut the edge in p with the largest ratio of eigenscore[5] to cost, since edges with high eigenscores are known to be important in network flow [18]. This version of the algorithm is called GreedyEigenscore. In both cases, edges from p^* are not allowed to be cut.

4.2 Synthetic and Real Networks

Our experiments are on synthetic and real networks. All networks are undirected.

For the synthetic networks, we run five different random 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 (ER) and Barabási–Albert (BA) graphs, 2^{14}-node stochastic Kronecker graphs, 285×285 lattices, and 565-node complete graphs.

We use seven weighted and unweighted networks. The unweighted networks are Wikispeedia graph (WIKI) [21], Oregon autonomous system network (AS) [10], and Pennsylvania road network (PA-ROAD) [11]. The weighted networks are Central Chilean Power Grid (GRID) [9], Lawrence Berkeley National Laboratory network data (LBL), the Northeast US Road Network (NEUS), and the DBLP coauthorship graph (DBLP) [3]. The networks range from 444 edges on 347 nodes to over 8.3M edges on over 1.8M nodes, with average degree ranging from over

[5] The eigenscore of an edge is the product of the entries in the principal eigenvector of the adjacency matrix corresponding to the edge's vertices.

2.5 to over 46.5 nodes and number of triangles ranging from 40 to close to 27M. Further details on the real and synthetic networks—including URLs to the real data—are provided in the supplementary material.

For synthetic networks and unweighted real networks, we try three different edge-weight initialization schemes: Poisson, uniform random, or equal weights. 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.

4.3 Experimental Setup

For each graph—considering graphs with different edge-weighting schemes as distinct—we run 100 experiments unless otherwise noted. For each graph, 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 50 hops away from s[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).

The experiments were run on Linux machines with 32 cores and 192 GB of memory. The LP in `PATHATTACK-LP` was implemented using Gurobi 9.1.1, and shortest paths were computed using `shortest_simple_paths` in NetworkX.[7]

4.4 Results

Across over 20,000 experiments, `PATHATTACK-LP` finds the optimal solution (where the relaxed LP yields only integers) in over 98% of cases. In addition, the number of constraints used by `PATHATTACK` is typically a small fraction of the number of edges (M): at most 5% of M in 99% of our experiments. For brevity, we highlight a few results in this section. See the supplementary material for more results on each network and weighting scheme.

We treat the result of `GreedyCost` as our baseline cost and report the cost of other algorithms' solutions as a reduction from the baseline. With one exception[8], `GreedyCost` outperforms `GreedyEigenscore` in both running time and edge removal cost, so we omit the `GreedyEigenscore` results for clarity of presentation. Fig. 3 shows the results on synthetic networks, Fig. 4 shows the results on real networks with synthetic edge weights, and Fig. 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.

[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.

[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`.

[8] `GreedyEigenscore` only outperforms `GreedyCost` in COMP with uniform weights.

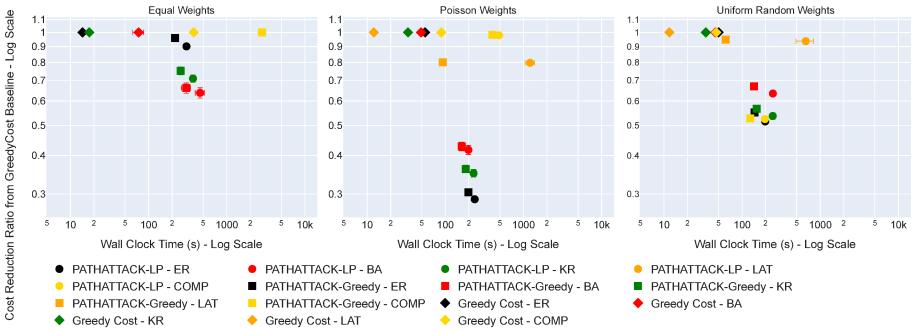

Fig. 3. Results on synthetic networks. Shapes represent different algorithms and colors represent different networks. The horizontal axis represents wall clock time in seconds and the vertical axis represents edge removal cost as a proportion of the cost required by the `GreedyCost` baseline. Lower cost reduction ratio and lower wall clock time is better. `PATHATTACK` yields a substantial cost reduction for weighted ER, BA, and KR graphs, while the baseline achieves nearly optimal performance for LAT.

Comparing the cost achieved by `PATHATTACK` to those obtained by the greedy baseline, we observe some interesting phenomena. Across the synthetic networks in Fig. 3, the real graphs with synthetic weights in Fig. 4, and the graphs with real weights in Fig. 5, lattices and road networks 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-LP` typically results in the optimal solution, this means that the baselines are achieving 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.

Cliques (COMP, yellow in Fig. 3) 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 run time 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, 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

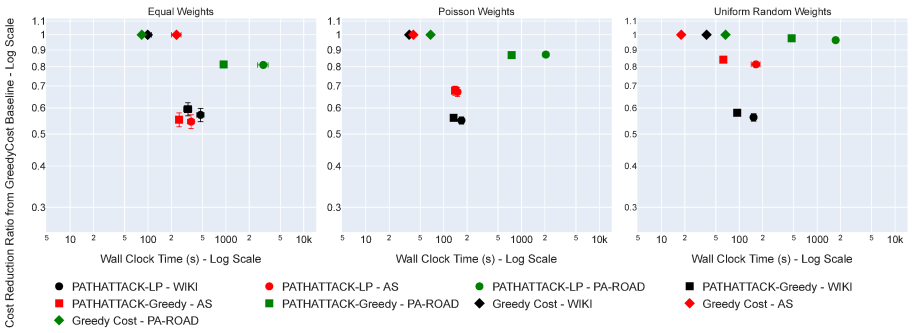

Fig. 4. Results on unweighted real networks. Shapes represent different algorithms and colors represent different networks. The horizontal axis represents wall clock time in seconds and the vertical axis represents edge removal cost as a proportion of the cost required by the `GreedyCost` baseline. Lower cost reduction ratio and lower wall clock time is better. As with synthetic networks, `PATHATTACK` significantly reduces cost in networks other than those that are grid-like, where the baseline is nearly optimal.

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 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 supplementary material). 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.

5 Related Work

Early work on attacking networks focused on disconnecting them [1]. This work demonstrated that targeted removal of high-degree nodes was highly effective against networks with powerlaw degree distributions (e.g., BA 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., [13]) attempts to efficiently find the single edge whose removal most increases the distance between two nodes. In contrast, we consider a devious adversary that wishes a certain path to be shortest.

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 [7]. The complement of this is work on network interdiction, where the goal is to intercept an adversary who is attempting to traverse the graph while remaining hidden. This problem has been

Fig. 5. Results on weighted real networks. Shapes represent different algorithms and colors represent different networks. The horizontal axis represents wall clock time in seconds and the vertical axis represents edge removal cost as a proportion of the cost required by the `GreedyCost` baseline. Lower cost reduction ratio and lower wall clock time is better. `PATHATTACK` reduces the cost of attacking the DBLP social network, while the other networks (those with low clustering) achieve high performance with the baselines. Note: the range of the time axis is lower than that of the previous plots.

studied in a game theoretic context for many years [20], and has expanded into work on disrupting attacks, with the graph representing an attack plan [12]. In this work, as in ours, oracles can be used to avoid enumerating an exponentially large number of possible strategies [6].

Work on Stackelberg planning [17] is also relevant, though somewhat distinct from our problem. This work adopts a leader-follower paradigm, where rather than forcing the follower to make a specific set of actions, the leader's goal is to make whatever action the follower takes as costly as possible. This could be placed in our context by having the leader (adversary) attempt to make the follower take the longest path possible between the source and the destination, though finding this path would be NP-hard in general.

Another related area is the common use of heuristics, such as using Euclidean distances to approximate graph distances [16]. Exploiting deviations in the heuristic enables an adversary to manipulate automated plans. Fuzzy matching has been used to quickly solve large-scale problems [15]. Attacks and defenses in this context is an interesting area for inquiry. A problem similar to Stackelberg planning is the adversarial stochastic shortest path problem, where the goal is to maximize reward while traversing over a highly uncertain state space [14].

There has recently been a great deal of work on attacking machine learning methods where graphs are part of the input. Attacks against vertex classification [22, 23] and node embeddings [4] 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 commu-

nity detection algorithm [8]. Our work complements these efforts, expanding the space of adversarial graph analysis into another important graph mining task.

6 Conclusions

We introduce the Force Path Cut problem, in which an adversary's aim is to force a specified path to be the shortest between its endpoints by cutting edges within a required budget. Many real-world applications use shortest-path algorithms (e.g., routing problems in computer, power, road, or shipping networks). We show that an adversary can manipulate the network for his strategic advantage. While Force Path Cut is NP-complete, we show how it can be translated into Weighted Set Cover, thus enabling the use of established approximation algorithms to optimize cost within a logarithmic factor of the true optimum. With this insight, we propose the `PATHATTACK` algorithm, which uses a natural oracle to generate only those constraints needed to execute the approximation algorithms. Across various synthetic and real networks, we find that the `PATHATTACK-LP` variant identifies the optimal solution in over 98% of more than 20,000 randomized experiments. Another variant, `PATHATTACK-Greedy`, has very similar performance and typically runs faster than `PATHATTACK-LP`, while a greedy baseline method is faster still but with much higher cost.

Ethical Implications: This work demonstrates how an adversary can attack shortest paths in complex networks. Appropriate defenses include building resilient network structures (e.g., adding redundancy to form cliques around key communication channels) and developing methods that not only detect attacks, but also identify the most likely source of the attack (e.g., whether an edge failed due to a random outage or a malicious destruction).

Acknowledgments

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.

References

1. Albert, R., Jeong, H., Barabási, A.L.: Error and attack tolerance of complex networks. Nature **406**(6794), 378–382 (2000)

2. Ben-Ameur, W., Neto, J.: A constraint generation algorithm for large scale linear programs using multiple-points separation. Math. Program. **107**(3), 517–537 (2006)
3. Benson, A.R., Abebe, R., Schaub, M.T., Jadbabaie, A., Kleinberg, J.: Simplicial closure and higher-order link prediction. Proc. Nat. Acad. Sci. **115**(48), E11221–E11230 (2018)
4. Bojchevski, A., Günnemann, S.: Adversarial attacks on node embeddings via graph poisoning. In: ICML. pp. 695–704 (2019)
5. Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D., Yannakakis, M.: The complexity of multiterminal cuts. SIAM J. Comput. **23**(4), 864–894 (1994)
6. Jain, M., Korzhyk, D., Vaněk, O., Conitzer, V., Pěchouček, M., Tambe, M.: A double oracle algorithm for zero-sum security games on graphs. In: AAMAS. pp. 327–334 (2011)
7. Jun, M., D'Andrea, R.: Path planning for unmanned aerial vehicles in uncertain and adversarial environments. In: Cooperative Control: Models, Applications and Algorithms, pp. 95–110. Springer (2003)
8. Kegelmeyer, W.P., Wendt, J.D., Pinar, A.: An example of counter-adversarial community detection analysis. Tech. Rep. SAND2018-12068, Sandia National Laboratories (2018). https://doi.org/10.2172/1481570
9. Kim, H., Olave-Rojas, D., Álvarez-Miranda, E., Son, S.W.: In-depth data on the network structure and hourly activity of the central Chilean power grid. Sci. Data **5**(1), 1–10 (2018)
10. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: Densification laws, shrinking diameters and possible explanations. In: KDD. pp. 177–187 (2005)
11. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics **6**(1), 29–123 (2009)
12. Letchford, J., Vorobeychik, Y.: Optimal interdiction of attack plans. In: AAMAS. p. 199–206 (2013)
13. Nardelli, E., Proietti, G., Widmayer, P.: Finding the most vital node of a shortest path. Theoretical Computer Science **296**(1), 167–177 (2003)
14. Neu, G., Gyorgy, A., Szepesvari, C.: The adversarial stochastic shortest path problem with unknown transition probabilities. In: AISTATS. pp. 805–813 (2012)
15. Qiao, M., Cheng, H., Yu, J.X.: Querying shortest path distance with bounded errors in large graphs. In: SSDBM. pp. 255–273 (2011)
16. Rayner, D.C., Bowling, M., Sturtevant, N.: Euclidean heuristic optimization. In: AAAI (2011)
17. Speicher, P., Steinmetz, M., Backes, M., Hoffmann, J., Künnemann, R.: Stackelberg planning: Towards effective leader-follower state space search. In: AAAI. pp. 6286–6293 (2018)
18. Tong, H., Prakash, B.A., Eliassi-Rad, T., Faloutsos, M., Faloutsos, C.: Gelling, and melting, large graphs by edge manipulation. In: CIKM. pp. 245–254 (2012)
19. Vazirani, V.V.: Approximation Algorithms. Springer (2003)
20. Washburn, A., Wood, K.: Two-person zero-sum games for network interdiction. Operations Research **43**(2), 243–251 (1995)
21. West, R., Pineau, J., Precup, D.: Wikispeedia: An online game for inferring semantic distances between concepts. In: IJCAI (2009)
22. Zügner, D., Akbarnejad, A., Günnemann, S.: Adversarial attacks on neural networks for graph data. In: KDD. pp. 2847–2856 (2018)
23. Zügner, D., Günnemann, S.: Certifiable robustness and robust training for graph convolutional networks. In: KDD. pp. 246–256 (2019)