Practical Anytime Algorithms for Judicious Partitioning of Active Directory Attack Graphs *

Yumeng Zhang[1] , **Max Ward**[2] and **Hung Nguyen**[1]
[1]The University of Adelaide
[2]University of Western Australia
{yumeng.zhang01, hung.nguyen}@adelaide.edu.au, max.ward@uwa.edu.au

Abstract

Given a directed graph, a set of source nodes, a target node and a budget, we study the problem of maximizing the number of source nodes disconnected from the target node by removing edges not exceeding the budget. Our model is mainly motivated by a cyber security use case where we need to minimize the attack surface of a Windows Active Directory system. In these high-profile attacks, the attackers first compromise a source (i.e., a compromised user node) and then laterally move to a destination (i.e., a high-privileged admin node). Our aim is to minimize the number of users with a path to the admin.

We first prove that the problem is NP-hard. Algorithms for exact optimality usually struggle to converge on graphs that approach real-world network scales and therefore are not practical for usage. In light of this, we study anytime algorithms that return an acceptable result whenever the algorithm is terminated, and can improve optimality by allowing longer computational time. We observe the source connectivity of directed graphs, based on which we propose a novel anytime algorithm—the spiral algorithm. We also develop two Monte Carlo Tree Search (MCTS) algorithms as a baseline to study the performance of typical anytime algorithms for our problem, and show that the spiral algorithm improves the optimality at a significantly faster speed and therefore exhibits better anytime behavior compared with MCTS.

1 Introduction

We model current problems in cyber security with judicious partitioning of directed graphs. We start by describing the cyber security problems that motivate our model and design rationales. Then, we give effective algorithms to solve them. Our primary contributions are new theoretical models and algorithms. Our design choices are heavily influenced by the cyber security use case and the urgent demand for practical solutions to these problems by security teams.

Microsoft Active Directory (AD) is the default security management system for Windows domain networks and is used by approximately 90% of Global Fortune 1000 companies [Krishnamoorthi and Carleton, 2020]. An AD environment is naturally described as a graph where the nodes are accounts/computers/groups, and the directed edges represent accesses. Minimizing attack paths in an AD system is a fundamental problem in AD security. There are many open-source and commercial tools for analyzing AD attack graphs. BloodHound by SpecterOps [Robbins *et al.*, 2016] is a popular AD analysis tool that is able to enumerate attack paths that an attacker can follow from a source node to the admin node. BloodHound Enterprise also provides actionable fixes in the form of a set of edges (accesses) that should be removed to reduce the number of attack paths. Unfortunately, tools like BloodHound focus on removing some shortest attack paths but not all attack paths from a given entry node. As such, these solutions do not guarantee that no path remains between an entry node and a high-value target even if enumerated shortest paths from that entry node are eliminated. It leaves the possibility of an attack even after the clean-up. Removing an edge in an AD graph is costly in practice as every edge removal is manually examined and approved to make sure it does not disrupt normal operations and cause cascade effects. These reviews often take days or weeks and are performed by IT operational teams instead of the security team (who uses Bloodhound).

The goal of an AD defender is to eliminate attack paths by removing a limited number of connections. We formalize this problem as finding a budget-constrained set of edges to remove such that the number of attacker entry points is minimized. First, we observe that this problem is new to graph theory, but it falls into a class of problems called *judicious partitioning*. We prove that our problem, like other judicious partitioning problems, is NP-hard. For most cyber security use cases, solutions do not need to be perfect, only good and fast enough for timely remediation actions. Based on this observation, we develop the first practical algorithm for this judicious partitioning formulation of the problem. It is an anytime algorithm that significantly outperforms Monte Carlo Tree Search (MCTS) and integer linear programming (ILP).

*Source code and data are available at: https://github.com/YMZhang7/ijcai24_judicious_partition

Our key contributions are:

- A new graph theoretical and combinatorial optimization problem under the class of judicious graph partitioning. It is a rarely studied problem. We motivate the problem through a practical use case of hardening AD systems.
- We prove that the problem is NP-hard and develop a set of anytime algorithms instead of traditional algorithms for balancing the trade-off between the timeliness and the optimality of a solution.
- A new graph connectivity concept—source connectivity. We adopt source connectivity for an ordering of subgraphs, which is the key enabler for our algorithms.
- A novel anytime algorithm: the spiral algorithm. The algorithm iteratively runs several sub-algorithms; a solution can be found at the end of each iteration. The computation progresses spirally and the optimality of the solution monotonically improves over iterations.

Our new algorithm is able to converge on optimal solutions within minutes on large AD graphs. In contrast, an optimized MCTS produces solutions that are an order of magnitude worse after an hour. Our algorithm is also orders of magnitude faster than solving the problem directly using ILP. Our algorithm is the only one able to find good solutions on AD networks of practical sizes.

2 Problem Definition and Related Work

2.1 Problem Definition

Problem 1. Given a directed graph $G = (V, E)$, a set of source nodes $S = \{s_1, s_2, \ldots, s_k\}$, $S \subseteq V$, $|S| = k$ and one sink node $t \in V$, $t \notin S$. A cut $c = \{(u, v) \in E : u \in V_1, v \in V_2\}$ partitions V into V_1 and V_2, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. Given an integer budget b, the objective is to find an optimal c^* for a maximum $S^* = S \cap V_1$ that satisfies $|c^*| \leq b$.

Theorem 1. *Problem 1 is NP-hard.*

Proof. We prove that our problem is NP-hard by reducing from the k-cluster problem. Given an undirected graph, a k-cluster is defined as a subgraph with k vertices that has the number of edges maximized. The k-cluster problem has been proven to be NP-complete [Corneil and Perl, 1984].

Suppose there exists an undirected graph $G' = (V', E')$. We construct a directed graph $G = (V, E)$ by first creating one node s_e for each edge $e \in E'$, and we call this set of nodes V_a. Next, we create a node i_v for each node $v \in V'$ and denote this set of nodes as V_b. Lastly, we add a target node t. Combining these nodes, we have $V = V_a \cup V_b \cup \{t\}$. Now, for each edge $e = (u, v) \in E'$, we direct an edge in G from s_e to i_u and another edge from s_e to i_v, and we use E_1 to denote this set of edges. Then, we direct an edge from i_v to t for each $v \in V'$ and call this set of edges E_2. In this way, $E = E_1 \cup E_2$. We complete the construction of G based on G'. An example is given in Figure 1.

We define V_a to be the set of source nodes. We continue using c to denote the cut that partitions V into V_1 and V_2, and S^* as a maximum $S \cap V_1$. In the directed graph G, it is apparent that we can always find an optimal c for a maximum S^* from E_2. The identified c can be assumed to contain only edges from E_2 without loss of generality. Even if c contains edges from E_1, they can be replaced by edges from E_2 without affecting $|S^*|$. Suppose there were a solution to Problem 1. We could find a cut c consisting of at most b edges in E_2 to remove that would maximize $|S^*|$. This c would simultaneously cut at most b nodes in V_2. This corresponds to b nodes in the undirected graph G' and edges incident on the b nodes being maximized, which is equivalent to solving the k-cluster problem. As the k-cluster problem is NP-complete, our problem of finding an optimal c for a maximum S^* would be NP-hard. □

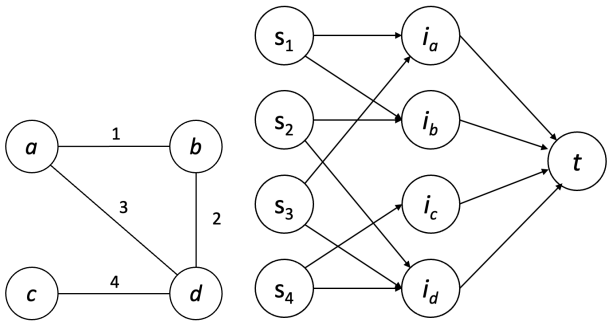

Figure 1: An example of constructing a directed graph given an undirected graph.

2.2 Related Work

Cyber attack graphs model the chain of events (conceptual or physical) that lead to successful cyber attacks. The term "attack graph" was first introduced around 1998 [Phillips and Swiler, 1998; Schneier, 1999; Ortalo *et al.*, 1999], and since then there have been more than 90 different definitions of attack graphs and methods to construct them, see for example: [Ritchey and Ammann, 2000; Sheyner *et al.*, 2002; Ou *et al.*, 2006; Barik *et al.*, 2016; Lallie *et al.*, 2020]. One of the most prominent and actively used attack graph models is the Active Directory (AD) attack graph that models identity-based snowball attacks in Windows systems, first proposed in [Dunagan *et al.*, 2009], developed further in [Bouillot and Gras, 2014] and then commercialized by the BloodHound tool [Robbins *et al.*, 2016]. In an AD attack graph, multifarious network entities including user accounts, computers, security groups, Organizational Units (OUs), and Group Policy Objects (GPOs) are modeled as nodes, and a myriad of exploitable interrelationships and dependencies are modeled as directed edges.

Being able to audit, identify, and remove attack paths in an AD environment is crucial for containing and limiting the impact of security breaches in an organization. There are many existing solutions for attack path management including the commercial BloodHound Enterprise [Robbins *et al.*, 2016]. BloodHound and similar defensive attack path management solutions rely heavily on the identification and removal of edges in the AD attack graphs. The underlying idea at the core of all those solutions is to find edges whose removals

will cut as many shortest attack paths as possible. These solutions are widely used in the cyber security industry but do not guarantee that no attack paths exist after the cleanup. We focus in this work on solutions that could completely remove *any* paths between an entry point and the high-value target. To date, there rarely exists literature that studies our problem. Here, we provide a review of problems that are related or similar to our problem.

Removing edges so that source nodes and the target node are divided into two disconnected subgraphs is related to the graph partitioning problem, which studies the problem of partitioning the graph into several vertex classes so that the sum of weights on vertices in each vertex class satisfies some constraints, or the sum of weights on edges across vertex classes is optimized [Fjällström, 1998]. It becomes a *bipartitioning problem* when the graph is to be divided into only two vertex classes. *Minimum cut* and *max cut* problems are two classic graph bipartitioning problems. The minimum cut problem is about finding a cut such that a source node and a sink node are in different vertex classes, and the sum of weights on edges in the cut is minimized. The minimum cut problem is polynomial-time solvable based on the max-flow min-cut theorem [Korte and Vygen, 2012]. In the max cut problem, on the other hand, the goal is to find a cut with a total weight larger than a specified value. This is a well-established NP-hard problem [Garey *et al.*, 1976].

In our problem, not only is the size of the cut being optimized, but we also expect the number of nodes from a particular set to be maximized in a vertex class after the partition. When there are several parameters to be optimized simultaneously, it becomes a *judicious partitioning problem* [Scott, 2005; Bollobás and Scott, 2002; Lee *et al.*, 2016]. A typical judicious partitioning problem is an extension of the max cut problem. When the number of edges between vertex classes is maximized, it can be expected that the number of edges in each vertex class is reduced simultaneously. The judicious partitioning problem derived from this is the minimization of the number of edges in each vertex class. The majority of past research focused on the extremal study of the problem, which can provide information for the development and analysis of algorithms. Additional constraints on the graph like graph degree are usually added for the derivation of the bound [Bollobás and Scott, 2004]. However, there hardly exists studies on efficient algorithms for solving the judicious partitioning problem, and having the maximization of source nodes in a different vertex class from the target node as the second optimization objective is also a rarely explored subbranch under the judicious partitioning category.

Another branch of study closely related to this problem is the maximum flow interdiction problem. It is about removing a limited number of edges to minimize the maximum flow between a source and a sink, which is equivalent to minimizing the minimum cut because of the max-flow min-cut theorem. The max flow interdiction problem has been proven to be strongly NP-hard for non-planar networks [Wood, 1993]. Furthermore, reducing the minimum cut cannot provide a reduction in the number of sources and therefore cannot solve our problem.

Despite the connections these problems share with our problem, the majority of them are NP-hard. More importantly, existing solutions to those problems are not directly applicable to our problem. A thorough investigation of our problem and efficient solutions remains a significant gap.

3 Spiral—A Practical Anytime Algorithm for Judicious Partitioning of Graphs

3.1 Algorithm Design

The basic structure of our algorithm is to iteratively improve a compressed set of variables created to solve Problem 1. As we will show, several insights about the problem are required to formulate the algorithm. Note that our algorithm is anytime, which means it can produce a solution whenever it is interrupted. This makes it practical in real-world situations.

We first start with an Integer Linear Programming (ILP) approach to find the maximum number of source nodes that could be disconnected by removing a limited number of edges. Note that ILP algorithms do not guarantee polynomial time termination for judicious partitioning. Our ILP formulation is similar to [Ngo *et al.*, 2024] and works on edges instead of nodes that are to be removed for the maximization of $|S^*|$. Our ILP formulation is provided in Equation (1).

$$
\begin{aligned}
\max \quad & |S| - \sum_{s \in S} var(s) \\
\text{s.t.} \quad & var(u) \geq var(v) - var((u,v)), && \forall (u,v) \in E \\
& \sum_{(u,v) \in E} var((u,v)) \leq b, \\
& var((u,v)) \in \{0,1\}, && \forall (u,v) \in E \\
& var(t) = 1, \\
& var(v) \in \{0,1\}, && \forall v \in V - \{t\}
\end{aligned}
\tag{1}
$$

In this ILP, we create binary variables for all nodes and edges in G. If an edge (u, v) is cut, its corresponding variable $var((u, v))$ becomes 1. Conversely, if a node u cannot reach t, its variable $var(u)$ becomes 0. $var(t)$ should be fixed at 1. A node u can reach t as long as one of its outgoing edges leads to t. The outgoing edge (u, v) can connect u to t if v can reach t and $var((u, v)) = 0$. The first constraint ensures that $var(u)$ becomes 1 if either one of u's outgoing edges (u, v) has $var(v) = 1$ and $var((u, v)) = 0$. The objective function models the maximization of disconnected source nodes. As an entry node's variable equals 1 if it can reach t, the sum of variables for source nodes is minimized for the maximization of $|S^*|$.

The computational efficiency of the ILP solver suffers as the network size expands. One way to improve the scalability of an ILP solution to an NP-hard problem is to contain the explosion of the number of variables associated with the input size. The concept of *source connectivity* can help us achieve that. We explain this novel concept in the next section.

3.2 Source Connectivity

Definition 3.1 (k-source connectivity)**.** Given a directed graph G, a set of source nodes S and a target node t, we define G to be k-source-connected if a total of k source nodes have at least one path to t.

We can order the subgraphs of G in terms of source connectivity. Suppose there are two subgraphs G'_1 and G'_2 in G. If the number of source nodes that have a path to t in G'_1 is larger than that in G'_2, we say that G'_1 has larger source connectivity than G'_2. In this way, subgraphs of a graph G can be ordered in decreasing order of source connectivity.

We say that a k-source-connected subgraph is eliminated if at least one edge in that subgraph is cut: after the edge is removed from G, any k-source-connected subgraph we find from G must not have the removed edge and therefore cannot be the same. In this sense, finding an optimal cut to maximize disconnected source nodes S^* is equivalent to eliminating subgraphs in decreasing order of source connectivity so that the last k-source-connected subgraph we find has k minimized.

Based on this idea of subgraph elimination, we can see that a k-source-connected subgraph provides an edge space from which we find an optimal cut c^* to maximize $|S^*|$. This edge space can be further reduced if we scale down a k-source-connected subgraph by allowing only one path from each of the k source nodes to t. In this way, it becomes a k-source-connected tree. In the following discussions, a k-source-connected tree will be referred to as a k-tree and denoted as T_k.

To sample a T_k, we adopt Dijkstra's algorithm to find a shortest (s,t)-path for k connected $s \in S$. The path length is calculated as the number of edges so that the number of edges in a T_k can be further reduced.

Although Dijkstra's algorithm is typically adopted for finding one shortest (s,t)-path, we can easily adapt it so that one shortest path from each connected node to t is recorded in one traversal. We can then construct a k-tree with shortest paths between k source nodes and t. We use $T(G,S,t,k)$ to denote this procedure.

3.3 Using Source Connectivity to Solve Judicious Partitioning

Suppose $b = 1$ and initially all nodes in S have at least one path to t in G. If $S^* \subseteq S$ can be disconnected by removing one edge, then any $|S|$-tree must contain this edge: if a node $s \in S^*$ is cut, all of its paths must be cut, any one of the (s,t)-paths must have that one edge as c^*, including the path in the enumerated $|S|$-tree. We can see that the problem is actually polynomial-time solvable when $b = 1$.

However, according to Theorem 1, the problem becomes an NP-hard combinatorial optimization problem when we allow more than one edge in c^*. Different combinations of edges need to be evaluated as a cut to find the maximum S^*. Some source nodes may also require removing more than one edge to disconnect. Therefore, a greater number of trees with some specified source connectivity should be enumerated for a larger pool of edge candidates to find the optimal c^*. An enumerated tree may also have more than one edge to be added to c^*.

With the above observation, we can see that it is not necessary to create variables for all edges in G. An optimal c^* for maximizing S^* can be found in a small set of trees with certain source connectivity. We only need to enumerate a subset of trees in decreasing order of source connectivity and create variables for edges in those trees to give to ILP. We denote the set of edges found in the enumerated trees as E^*.

Edges not in E^*, which are not to be created as variables, can have values fixed at 0 in the ILP to model the fact that the edge is not considered for a cut and always allows a path. In the following discussion, we use ILP(E^*) to denote the ILP running with edge candidates E^*. ILP(E^*) returns the maximum $|S^*|$. We can also retrieve the optimal c^* for the maximized result after ILP converges.

3.4 Spiral Anytime Algorithm

We propose an anytime algorithm that maximizes S^* in a spiral pattern with the above observations. It has two sub-algorithms running alternately and cooperatively to move the optimality upward. ILP constitutes one sub-algorithm that is responsible for finding a cut c under the budget that maximizes S^* with edge candidates E^* that are provided. The other sub-algorithm limits the number of variables given to ILP by conservatively finding unexplored edges to add into E^*. An unexplored edge is an edge that is not in E^*.

The optimization process progresses incrementally in a layered manner. Each time, the algorithm focuses only on enumerating k-trees with the exact source connectivity of k to test whether $|S^*|$ can be optimized to $|S|-k+1$. We call this examination of k-trees a *layer* in the spiral optimization process. A layer contains several k-tree enumeration iterations. If ILP(E^*) $\leq |S| - k$ and there exists a k-tree that has at least one unexplored edge, the optimization cannot progress to the next layer because there exist some k-trees that are not eliminated.

If ILP(E^*) $> |S| - k$, the algorithm progresses to the next layer and starts examining $(|S| - \text{ILP}(E^*) - 1)$-trees. E^* can be cleared at the start of the new layer to reduce the space of edges being considered and improve the speed of ILP.

If $|S^*|$ is the maximum number of source nodes that the budget can cut, it implies that the budget cannot eliminate all $(|S| - |S^*|)$-trees, therefore the examination of $(|S| - |S^*|)$-trees would be the last layer before termination. To terminate, the algorithm would eventually need to add all edges from E to E^*.

With the above-mentioned process, the algorithm iteratively improves c and has the number of disconnected source nodes, namely $|S^*|$, monotonically increase over time.

Proposition 2. *If ILP(E^*) $> |S| - k$, then the converged c eliminates all k-trees even if $|E^*| < |E|$.*

Proof. Even though only edges in E^* are created as variables in ILP, ILP is running on the complete graph where edges that are not created as variables invariably allow a path. If there exists a k-tree that is not eliminated, k source nodes must remain connected to t and the ILP would return $|S| - k$. □

Proposition 3. *If all k-trees can be eliminated under budget, all subgraphs with source connectivity larger than k are also eliminated using the same cut.*

Proof. Larger trees contain smaller trees as subtrees. That is, every k-tree is contained inside a $(k + x)$-tree, $x > 0$. If all k-trees can be eliminated under budget, then all $(k + x)$-trees would be eliminated simultaneously. Even when the

examination of k-trees is the last layer before termination, the algorithm would find edges for cutting all $(k + x)$-trees, as the algorithm would not have progressed to examining k-trees otherwise. □

The pseudocode for the algorithm is given in Algorithm 1. In Line 9, ILP provides us with the maximized number of disconnected source nodes $|S^*| = x$, and the cut c that achieves this result given the edge candidates E^*. In Line 10, x is used to determine whether the algorithm progresses to the next layer and starts examining trees with smaller source connectivity. The latest c that gives the largest max_disconnected would be returned when the algorithm is interrupted. Otherwise, the algorithm terminates of its own accord when all edges in E are added into E^*.

Algorithm 1 Spiral anytime algorithm

Input: $G = \{V, E\}, S, t, b$

```
1: E* ← {}
2: max_disconnected ← 0
3: c ← ∅
4: while within time limit do
5:     if E* = E then
6:         return c
7:     k = |S| − max_disconnected − 1
8:     E* ← E* ∪ edge_sampling(k)
9:     (x, c) ← ILP(E*)
10:    if x > max_disconnected then
11:        max_disconnected ← x
12:        E* ← {}
13: return c
```

In Algorithm 1, edge_sampling(k) finds a k-tree that has unexplored edges. Ideally, the k-trees sampled over iterations should have a consistent amount of unexplored edges so that E^* expands steadily, and the anytime algorithm makes gradual headway. However, this problem is NP-hard even for sampling a simple path with only one unexplored edge.

Problem 2. Given a directed graph $G = (V, E)$, a set of source nodes S, a target node t, a constant k, and a set of edges $E^* \subseteq E$. The objective is to find a k-source-connected tree from G that has at least one edge in $E - E^*$.

Proposition 4. *Problem 2 is NP-hard.*

Proof. First, we prove that finding a *simple path* that goes through a specific edge is NP-hard. It can be proven by reducing from the *2 Edge-disjoint Paths* problem [Tholey, 2006].

In the 2 edge-disjoint paths problem, we are given four distinct nodes s, u, v, t and we determine whether there exist two edge-disjoint paths, P_1 from s to u and P_2 from v to t.

Assume for contradiction that there exists an algorithm A that can find a simple path from s to t that goes through a specific edge (u, v) in polynomial time. Now, for any given instance of the 2 edge-disjoint paths problem, add a new edge (u, v) to the graph. We can use A to find a path from s to t that goes through (u, v).

Algorithm A also solves the problem of finding 2 edge-disjoint paths P_1 and P_2, which means that the problem of finding a simple (s, t)-path that goes through edge (u, v) is at least as hard as the 2 edge-disjoint problem, and is therefore NP-hard.

We can reduce finding a simple (s, t)-path through a forced edge (u, v) to Problem 2. Construct an input over the original graph containing a single source $S = \{s\}$, a target t, a source connectivity of $k = 1$, and where $E - E^* = \{(u, v)\}$. This reduction is polynomial and concludes the proof that Problem 2 is NP-hard. □

To simplify the edge-sampling problem, we specify that the sampling algorithm only needs to try its best at sampling edges that can form a k-tree, and the sampled edges only need to include at least one unexplored edge. We adopt Dijkstra's algorithm as part of this approximate edge-sampling algorithm. It is the same procedure as $T(G, S, t, k)$ given in Section 3.2. However, we calculate the path length with edge weights instead of the number of edges for the sampling of unexplored edges. Here, we use $T'(G, S, t, k)$ to denote this procedure.

We use $w(e)$ to denote the weight value on edge e. All unexplored edges should have a weight value of 0. At the beginning of each layer, we set $w(e) = 0$ for all edges in E. After an unexplored edge e is sampled by edge_sampling(k), we increment $w(e)$ by 1. The path length is calculated in terms of the sum of edge weights so that edges with zero weights, namely unexplored edges, are more likely to be sampled. However, zero-weight edges are not invariably on shortest paths. As we have established in Problem 2, finding a k-tree that has at least one unexplored edge is NP-hard. When Dijkstra's algorithm cannot find zero-weight edges, we randomly sample z unexplored edges from E so that the algorithm is guaranteed to make progress in expanding E^* in each iteration. The pseudocode can be found in Algorithm 2.

Algorithm 2 Dijkstra-based edge sampling

Input: $G = \{V, E\}, S, t, b$

Require: At the start of each layer in the spiral algorithm, set $w(e) = 0$ for all $e \in E$

```
1: function EDGE_SAMPLING(k)
2:     unexplored ← {}
3:     if ∃e ∈ E : w(e) = 0 then
4:         T_k = T'(G, S, t, k)
5:         for e ∈ T_k do
6:             if w(e) = 0 then
7:                 unexplored.add(e)
8:         if unexplored = ∅ then
9:             unexplored ← uniform_sample(E − E*, z)
10:        for e ∈ unexplored do
11:            w(e) ← w(e) + 1
12:    return unexplored
```

4 Experimental Results

We test the algorithms on four simulated AD networks of different sizes, all of which are generated with ADSimulator[1].

[1]https://github.com/nicolas-carolo/adsimulator

Information on the sizes of the graphs is given in Table 1. The source nodes S consist of low-privileged user accounts and computers in the AD network, which are at higher risk of becoming attack entries in a cyber attack. The target node t is selected to be DomainAdmins—a security group managing high-privileged admin users that can give the hacker full control of the system once compromised.

GraphID	Nodes	Edges	Source Nodes
G_1	3191	28565	505
G_2	6191	57831	1091
G_3	12191	123583	2061
G_4	30191	347370	4868

Table 1: Graph instances for the experiments

4.1 Baseline—MCTS and ILP

We developed two baseline MCTS-based algorithms for comparison. The first algorithm (MCTS1) is a straightforward adaptation of MCTS for judicious partitioning, in which each state represents a set of edges c evaluated as a cut under budget, and a state transitions to another by sampling an additional edge from E to add to c. Initially, $c = \emptyset$. c represents a termination state if $|c| = b$. The second algorithm (MCTS2) is a significant modification of MCTS1. It adopts source connectivity for reducing the action space in each state. Both algorithms use the *Upper Confidence Bound* [Auer *et al.*, 2002] for balancing the exploitation of a promising set of edges and the exploration of other sets.

We also implemented the pure ILP solution provided in Equation (1). It runs on the complete graph where all nodes and edges are created as variables.

4.2 Results

We first examine the algorithms' performance at improving $|S^*|$ within some time limit. For this experiment, we only test the algorithm's performance in G_1 and G_4, with a budget of 5 and 20. We set various time limits spanning from 60s to 18000s, and run each algorithm 100 times. The recorded results are displayed in Figure 2, from which we can see that MCTS is relatively slow at improving $|S^*|$; even MCTS2 cannot give distinctive improvement to the performance. One potential reason could be that the search space in our problem is too big that it is difficult for MCTS algorithms to make any meaningful progress. On the other hand, the spiral anytime algorithm improves the optimality of the result at a much faster speed.

Next, we show a comparison between the pure ILP and the spiral algorithm in Table 2. ILP is capable of finding c^* with exact optimality. However, it struggles to converge as the graph size expands. To evaluate the spiral algorithm, we take the optimal $|S^*|$ returned by ILP and record the amount of time elapsed for the spiral anytime algorithm to reach the same $|S^*|$. Results suggest that the spiral algorithm can rapidly reach the optimal $|S^*|$, and its advantage over ILP becomes distinctive in larger graphs.

For problem instances where ILP cannot converge under one hour, we tested how the spiral algorithm improves $|S^*|$ in the course of one hour. The results are given in Figure 3, which provides more evidence that the algorithm can bring a great jump in optimality rapidly within seconds. Compared with ILP which cannot provide a solution until it terminates of its own accord, the spiral algorithm can be more useful in resolving urgent security issues in the real world where having a satisfactory solution promptly is more important than guaranteed optimality.

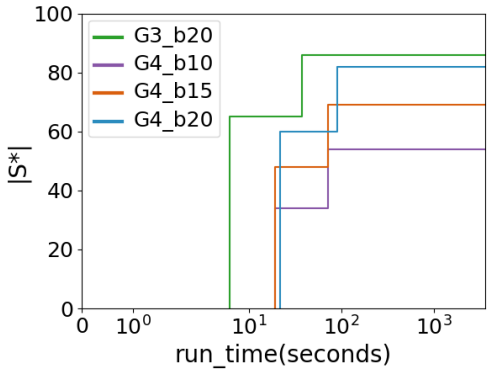

Figure 3: For problem instances in Table 2 where ILP cannot find a solution under time limit, spiral anytime algorithm makes great strides at improving $|S^*|$ within seconds.

In the above experiments, the spiral algorithm exhibits a high level of diminishing returns: the improvement in solution quality improves rapidly in the early stage and reaches stagnation until termination. This is a desirable quality in an anytime algorithm. For absolute optimality, the spiral algorithm needs to finish the last layer by eventually adding all edges into E^*. The last layer would take the longest time. The speed at which E^* expands affects the run time of the last layer: the slower the expansion speed is, the more iterations ILP needs to run. As E^* gets larger, the ILP's speed deteriorates and can severely impact the run time of each iteration. Nevertheless, when the algorithm stalls at a particular $|S^*|$ for a long time, there is a high chance that the algorithm has arrived at the optimal result, and our confidence in the optimality of the result strengthens as the run time extends.

Compared with MCTS, because of the spiral algorithm's capability to make fast improvements in optimality, it provides security-hardening suggestions of higher quality under the same run-time budget and therefore more helpful in resolving a security crisis.

The hardening of AD security has been studied as the shortest path interdiction problem in the past [Guo *et al.*, 2022; Guo *et al.*, 2023; Zhang *et al.*, 2023], which is for maximizing the shortest path length by removing a limited number of edges under budget. The double oracle algorithm [Zhang *et al.*, 2023] is an efficient heuristic for solving the shortest path interdiction problem on AD attack graphs. In Table 3, we show a comparison of the number of disconnected source nodes under the same budget using the double oracle algo-

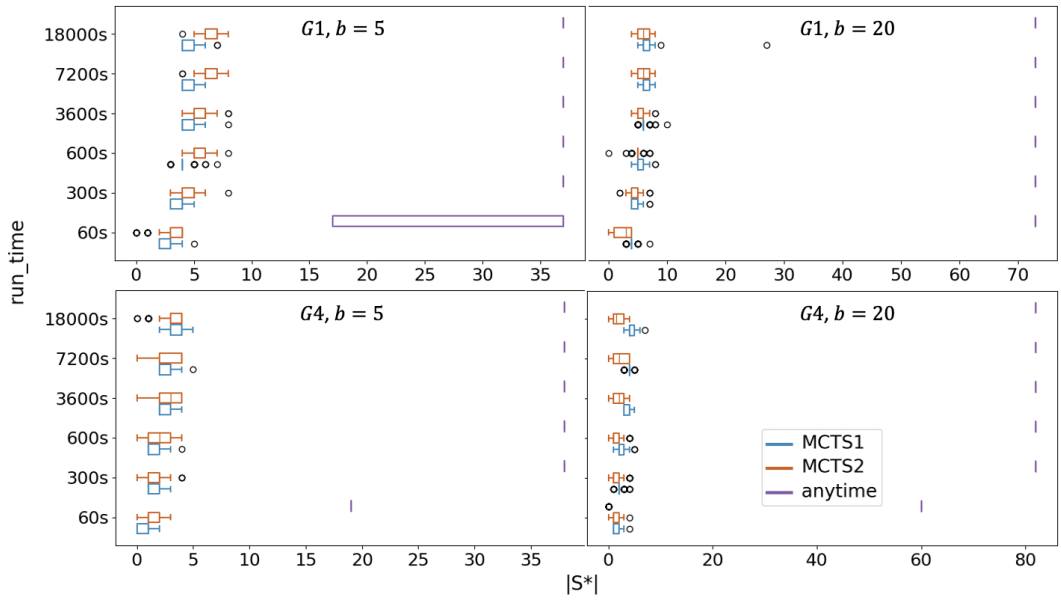

Figure 2: A comparison of the three algorithms' performance at improving $|S^*|$ within some time limit.

GraphID	$b = 5$		$b = 10$		$b = 15$		$b = 20$	
	ILP	Spiral	ILP	Spiral	ILP	Spiral	ILP	Spiral
G_1	1.89	17.70	15.50	10.87	19.61	19.78	54.11	19.57
G_2	23.79	23.41	116.53	25.73	431.28	33.04	382.09	33.14
G_3	87.76	33.87	143.69	40.60	1943.52	127.44	> 3600	-
G_4	1099.05	69.36	> 3600	-	> 3600	-	> 3600	-

Table 2: Run time (seconds) for optimal results. The spiral algorithm takes the optimal result given by ILP to record the amount of time elapsed for it to reach the same result.

rithm and the spiral algorithm. We allow 60 seconds of run time for the spiral algorithm, and specify a budget of 5 and 10 respectively. As we can see in the table, approaching the AD security-hardening problem as the shortest path interdiction problem cannot help the security engineer reduce the number of attack sources.

	$\|S\|$	Double Oracle $\|S^*\|$		Spiral $\|S^*\|$	
		$b = 5$	$b = 10$	$b = 5$	$b = 10$
G_1	505	9	18	37	52
G_2	1091	2	9	37	52
G_3	2061	14	21	42	58
G_4	4868	2	6	38	54

Table 3: A comparison of the number of disconnected sources between the spiral algorithm and the double oracle algorithm ($b = 5, 10$).

5 Conclusion

In this paper, we studied judicious partitioning of directed graphs as a security-hardening measure for Active Directory (AD) systems. We formulated the judicious partitioning problem in a novel way that is rarely studied in the past, a thorough investigation of our problem remains a significant gap. We first provided a proof that our problem is NP-hard. Optimal solutions are not practical for most AD graphs, therefore we studied anytime algorithms that are fast and provide good solutions for timely remediation actions. We proposed the spiral anytime algorithm that has a novel anytime optimization paradigm. Currently, it is the only practical algorithm available for our problem. Compared with the pure Integer Linear Programming solution and two Monte Carlo Tree Search solutions we investigated as the baseline, our spiral algorithm has significantly better performance in improving the optimality of the result on large AD graphs.

Acknowledgements

This work was supported with supercomputing resources provided by the Phoenix HPC service at the University of Adelaide.

References

[Auer *et al.*, 2002] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit

problem. *Machine learning*, 47:235–256, 2002.

[Barik *et al.*, 2016] Mridul Sankar Barik, Anirban Sengupta, and Chandan Mazumdar. Attack graph generation and analysis techniques. *Defence science journal*, 66(6):559, 2016.

[Bollobás and Scott, 2002] Béla Bollobás and Alex D Scott. Problems and results on judicious partitions. *Random Structures & Algorithms*, 21(3-4):414–430, 2002.

[Bollobás and Scott, 2004] Béla Bollobás and Alex D Scott. Judicious partitions of bounded-degree graphs. *Journal of Graph Theory*, 46(2):131–143, 2004.

[Bouillot and Gras, 2014] Lucas Bouillot and Emmanuel Gras. Chemins de contrôle en environnement active directory. *Proceedings of the SSTIC conférence 2014*, 2014.

[Corneil and Perl, 1984] Derek G Corneil and Yehoshua Perl. Clustering and domination in perfect graphs. *Discrete Applied Mathematics*, 9(1):27–39, 1984.

[Dunagan *et al.*, 2009] John Dunagan, Alice X Zheng, and Daniel R Simon. Heat-ray: combating identity snowball attacks using machinelearning, combinatorial optimization and attack graphs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 305–320, 2009.

[Fjällström, 1998] Per-Olof Fjällström. *Algorithms for graph partitioning: A survey*. Linköping University Electronic Press, 1998.

[Garey *et al.*, 1976] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[Guo *et al.*, 2022] Mingyu Guo, Jialiang Li, Aneta Neumann, Frank Neumann, and Hung Nguyen. Practical fixed-parameter algorithms for defending active directory style attack graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 9360–9367, 2022.

[Guo *et al.*, 2023] Mingyu Guo, Max Ward, Aneta Neumann, Frank Neumann, and Hung Nguyen. Scalable edge blocking algorithms for defending active directory style attack graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5649–5656, 2023.

[Korte and Vygen, 2012] Bernhard Korte and Jens Vygen. Network flows. *Combinatorial Optimization: Theory and Algorithms*, pages 173–209, 2012.

[Krishnamoorthi and Carleton, 2020] Swetha Krishnamoorthi and Jarad Carleton. Active directory holds the keys to your kingdom, but is it secure? https://www.frost.com/frost-perspectives/active-directory-holds-the-keys-to-your-kingdom-but-is-it-secure/, 2020.

[Lallie *et al.*, 2020] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, 35:100219, 2020.

[Lee *et al.*, 2016] Choongbum Lee, Po-Shen Loh, and Benny Sudakov. Judicious partitions of directed graphs. *Random Structures & Algorithms*, 48(1):147–170, 2016.

[Ngo *et al.*, 2024] Huy Quang Ngo, Mingyu Guo, and Hung Nguyen. Catch me if you can: Effective honeypot placement in dynamic ad attack graphs. *in Procs. of the IEEE Infocom, 2024*, 2024.

[Ortalo *et al.*, 1999] Rodolphe Ortalo, Yves Deswarte, and Mohamed Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, 1999.

[Ou *et al.*, 2006] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 336–345, New York, NY, USA, 2006. Association for Computing Machinery.

[Phillips and Swiler, 1998] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, page 71–79, New York, NY, USA, 1998. Association for Computing Machinery.

[Ritchey and Ammann, 2000] Ronald W Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 156–165. IEEE, 2000.

[Robbins *et al.*, 2016] Andy Robbins, Rohan Vazarkar, and Will Schroeder. Bloodhound: Six degrees of domain admin. https://bloodhound.readthedocs.io/en/latest/index.html, 2016. Accessed: 2022-12-14.

[Schneier, 1999] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[Scott, 2005] Alexander D Scott. Judicious partitions and related problems. In *BCC*, pages 95–117, 2005.

[Sheyner *et al.*, 2002] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, 2002.

[Tholey, 2006] Torsten Tholey. Solving the 2-disjoint paths problem in nearly linear time. *Theory of computing systems*, 39(1):51–78, 2006.

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

[Zhang *et al.*, 2023] Yumeng Zhang, Max Ward, Mingyu Guo, and Hung Nguyen. A scalable double oracle algorithm for hardening large active directory systems. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 993–1003, New York, NY, USA, 2023. Association for Computing Machinery.