Limited Query Graph Connectivity Test

Mingyu Guo, Jialiang Li, Aneta Neumann, Frank Neumann, Hung Nguyen

School of Computer and Mathematical Sciences, University of Adelaide, Australia
{mingyu.guo, j.li, aneta.neumann, frank.neumann, hung.nguyen}@adelaide.edu.au

Abstract

We propose a combinatorial optimisation model called *Limited Query Graph Connectivity Test*. We consider a graph whose edges have two possible states (ON/OFF). The edges' states are hidden initially. We could query an edge to reveal its state. Given a source s and a destination t, we aim to test $s-t$ connectivity by identifying either a *path* (consisting of only ON edges) or a *cut* (consisting of only OFF edges). We are limited to B queries, after which we stop regardless of whether graph connectivity is established. We aim to design a query policy that minimizes the expected number of queries.

Our model is mainly motivated by a cyber security use case where we need to establish whether attack paths exist in a given network, between a source (i.e., a compromised user node) and a destination (i.e., a high-privilege admin node). Edge query is resolved by manual effort from the IT admin, which is the motivation behind query minimization.

Our model is highly related to *Stochastic Boolean Function Evaluation (SBFE)*. There are two existing exact algorithms for SBFE that are prohibitively expensive. We propose a significantly more scalable exact algorithm. While previous exact algorithms only scale for trivial graphs (i.e., past works experimented on at most 20 edges), we empirically demonstrate that our algorithm is scalable for a wide range of much larger practical graphs (i.e., graphs representing Windows domain networks with tens of thousands of edges).

We also propose three heuristics. Our best-performing heuristic is via limiting the planning horizon of the exact algorithm. The other two are via reinforcement learning (RL) and Monte Carlo tree search (MCTS). We also derive an algorithm for computing the performance lower bound. Experimentally, we show that all our heuristics are near optimal. The heuristic building on the exact algorithm *outperforms all other heuristics*, surpassing RL, MCTS and eight existing heuristics ported from SBFE and related literature.

Introduction

Model Motivation

We propose a model called *Limited Query Graph Connectivity Test*, which is mainly motivated by a cyber security use case. We start by describing this use case to motivate our model and also to better explain the model design rationales. The main focus of this paper is the theoretical model and the algorithms behind. Nevertheless, our design choices are heavily influenced by the cyber security use case.

Microsoft Active Directory (AD) is the *default* security management system for Windows domain networks. An AD environment is naturally described as a graph where the nodes are accounts/computers/groups, and the *directed* edges represent *accesses*. There are many open source and commercial tools for analysing AD graphs. BLOODHOUND[1] is a popular AD analysis tool that is able to enumerate *attack paths* that an attacker can follow through from a source node to the admin node. IMPROHOUND[2] is another tool that is able to flag *tier violations* (i.e., this tool warns if there exist paths that originate from low-privilege nodes and reach high-privilege nodes). Existing tools like the above are able to identify attack paths, but they do not provide *directly implementable fixes*. In our context, a fix is a set of edges (accesses) that can be *safely* removed to eliminate all attack paths. Unfortunately, this cannot be found via *minimum cut*. Some edges may appear to be redundant but their removal could cause major disruptions (this is like refactoring legacy code – it takes effort). Consistent with industry practise, we assume that every edge removal is *manually* examined and approved, which is therefore costly. Our vision is a tool that acts like an "intelligent wizard" that guides the IT admin. In every step, the wizard would propose one edge to remove. The IT admin either approves it or rejects it. The wizard acts *adaptively* in the sense that past proposals and their results determine the next edge to propose. The wizard's goal is to minimize the expected number of proposals. *Ultimately, the goal is to save human effort during the algorithm-human collaboration*. We conclude the process if all attack paths have been eliminated, or we have established that elimination is impossible.[3] We also conclude if the number of proposals reaches a preset limit.

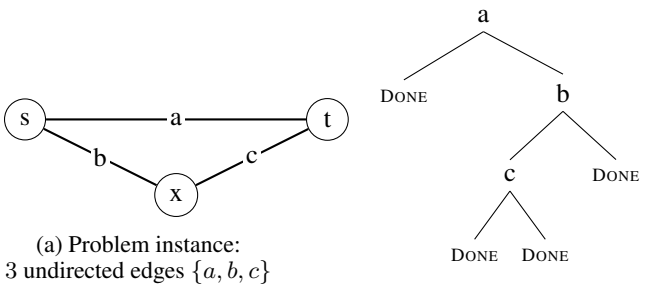
Formal Model Description

Definition 1 (Limited Query Graph Connectivity Test). The *Limited Query Graph Connectivity Test* problem in-

[1] https://github.com/BloodHoundAD/BloodHound

[2] https://github.com/improsec/ImproHound

[3] If we cannot eliminate all attack paths via access removal that are safe, then the IT admin needs to resort to more "invasive" defensive measures, such as banning accounts or splitting an account into two — one for daily usage and one for admin tasks.



(a) Problem instance: 3 undirected edges $\{a, b, c\}$ Edge is ON with $p = 0.5$ $B = 3$ (no limit)

(b) Optimal policy tree: Go left if query result is ON

Figure 1: *Limited Query Graph Connectivity Test* instance

volves a graph $G = (V, E)$, either undirected or directed. There is a single source $s \in V$ and a single destination $t \in V$. Every edge $e \in E$ has two possible states (ON/OFF). An edge is ON with probability p, where p is a constant model parameter. The edges' states are independent. An edge's state is hidden before we query it. By querying an edge, we reveal its state, which will stay revealed and never changes. We aim to design an *adaptive* query policy that specifies the query order (i.e., past queries and their results decide the next query). There are 3 terminating conditions:

- A *path certificate* is found: Among the already revealed edges, the ON edges form a *path* from s to t.
- A *cut certificate* is found: Among the already revealed edges, the OFF edges form a *cut* between s and t.
- We stop regardless after finishing B queries.

When there isn't ambiguity, we omit the word "certificate" and simply use *path*/*cut* to refer to path/cut certificate.

Our optimisation goal is to design a query policy that minimizes the expected number of queries spent before reaching termination. It should be noted that we are not searching for the shortest path or the minimum cut. Any certificate will do. If long paths/large cuts can be found using less queries, then we prefer them as our goal is solely to minimize the expected query count.

Example 1. *Let us consider the example instance as shown in Figure 1a. For this simple instance, the optimal query policy is given in Figure 1b, in the form of a binary policy tree. The first step is to query edge a in the root position. If the query result is* ON*, then we go left. Otherwise, we go right. The left child of a is* DONE*, which indicates that if a is* ON*, then the job is already done (a by itself forms a path). The right child of a is b, which indicates that if a is* OFF*, then we should query b next. If b's query result is* ON*, then we query c. Otherwise, we go right and reach* DONE *(both a and b being* OFF *form a cut). After querying c, we reach* DONE *regardless of c's state. If c is* ON*, then we have a path (b and c). If c is* OFF*, then we have a cut (a and c). The expected number of queries under this optimal policy is $1 + 0.5 + 0.25 = 1.75$ (we always query a; we query b with 50% chance; we query c with 25% chance).*

Related Models, Key Differences and Rationales

Our model is highly related to the *sequential testing* problem originally from the *operation research* community. An extensive survey can be found in (Ünlüyurt 2004). The sequential testing problem is best described via the following medical use case. Imagine that a doctor needs to diagnose a patient for a specific disease. There are 10 medical tests. Instead of applying all 10 tests at once, a cost-saving approach is to test one by one adaptively – finished tests and their results help pick the next test. Sequential testing has been applied to iron deficiency anemia diagnosis (Short and Domagalski 2013). (Yu et al. 2023) showed that 85% reduction of medical cost can be achieved via sequential testing. In the context of our model, the "tests" are the edge queries.

Our model is also highly related to *learning with attribute costs* in the context of *machine learning* (Sun, Chiu, and Cox 1996; Kaplan, Kushilevitz, and Mansour 2005; Golovin and Krause 2011). Here, the task is to construct a "cheap" classification tree (i.e., shallow decision tree), under the assumption that the features are costly to obtain. For example, at the root node, we examine only one feature and pay its cost. As we move down the classification tree, every node in the next/lower layer needs to examine a new feature, which is associated with an additional cost. The objective is to optimize for the cheapest tree by minimizing the expected total feature cost, weighted by the probabilities of different decision paths, subject to an accuracy threshold. In the context of our model, the "features" are the edge query results.

The last highly relevant model is *Stochastic Boolean Function Evaluation (SBFE)* (Allen et al. 2017; Deshpande, Hellerstein, and Kletenik 2014) from the *theoretical computer science* community. A SBFE instance involves a Boolean function f with multiple binary inputs and one binary output. The input bits are initially hidden and their values follow independent but not necessarily identical Bernoulli distributions. Each input bit has a query cost. The task is to query the input bits in an adaptive order, until there is enough information to determine the output of f. The goal is to minimize the expected query cost. In the context of our model, the "input bits" are the edges.

All the above models are similar with minor differences on technical details and they are all highly relevant to our cyber-motivated graph-focused model. Here, we highlight two key differences between our paper and past works.

1) We focus on *empirically scalable exact algorithms*.

There are two existing exact algorithms from SBFE literature, which are both prohibitively expensive. (Cox, Qiu, and Kuehner 1989)'s exact algorithm treats the problem as a MDP and uses the Bellman equation to calculate the value function for all problem states. Under our model, with m edges, the number of states is 3^m (an edge is hidden, ON, or OFF). (Allen et al. 2017) proposed an exact algorithm with a complexity of $O(n^{2^k})$, where k is either the number of paths or the number of cuts. *Existing works on exact algorithms only experimented on tiny instances with at most* 20 *edges* (Fu et al. 2017; Ben-dov 1981; Reinwald and Soland 1966; Breitbart and Reiter 1975).

Past results mostly focused on heuristics and approxima-

tion algorithms. (Ünlüyurt 2004)'s survey mentioned heuristics from (Jedrzejowicz 1983; Cox, Qiu, and Kuehner 1989; Sun, Chiu, and Cox 1996). Approximations algorithms were proposed in (Allen et al. 2017; Deshpande, Hellerstein, and Kletenik 2014; Kaplan, Kushilevitz, and Mansour 2005; Golovin and Krause 2011).

We argue that for many applications involving this lineage of models, *algorithm speed is not an important evaluation metric*, which is why we focus on empirically scalable exact algorithms. For example, for sequential medical tests, once a policy tree for diagnosing a specific disease is generated (however slow), the policy tree can be used for all future patients. In our experiments, we allocate 72 hours to the exact algorithm and we manage to exactly solve several fairly large graph instances, including one depicting a Windows domain network with 18795 edges. In comparison, going over 3^{18795} states is impossible if we apply the algorithm from (Cox, Qiu, and Kuehner 1989).

Incidentally, while deriving the exact algorithm, we obtain two valuable by-products: a lower bound algorithm and a heuristic that outperforms all existing heuristics from literature (we *ported* 8 heuristics to our model). The exact algorithm works by iteratively generating more and more expensive policy that eventually converges to the optimal policy. The lower bound algorithm is basically settling with the intermediate results. That is, if the exact algorithm does not scale (i.e., too slow to converge), then we stop at any point and whatever we have is a lower bound. Our best-performing heuristic also builds on the exact algorithm, which is via limiting the planning horizon of the exact algorithm.

2) We introduce a technical concept called *query limit*, which enables many positive results.

The query limit has a strong implication on scalability. Our exact algorithm builds on a few technical tricks for scalability, including a systematic way to identify the relevant edges that may be referenced by the optimal policy and a systematic way to generate the optimal policy tree structure. As mentioned earlier, without imposing a query limit, our algorithm is already capable of optimally solving several fairly large instances, including one with 18795 edges. Nevertheless, what really drives up scalability is the introduction of the query limit. Without the query limit, our exact algorithm is only scalable for special graphs. On the other hand, as long as the query limit is small, our exact algorithm scales. It should be noted that this paper is **not** on *fixed-parameter analysis* – our algorithm is **not** *fixed-parameter tractable* and the query limit is not the special parameter. The query limit is a technical tool for improving **empirical scalability**.

The query limit is a useful technical tool even for settings without query limit. Our lower bound algorithm is more scalable with smaller query limits. This is particularly nice because for settings without query limit, we could always artificially impose a query limit that is scalable and derive a performance lower bound, as imposing a limit never increases the query count. We are not aware of any prior work on lower bound for this lineage of models.

Our best-performing heuristic is also based on imposing an artificially small query limit on the exact algorithm (so that it scales – and we simply follow this limited-horizon exact algorithm). This way of constructing heuristic should be applicable for settings without query limit.

The query limit is often not a restriction when it comes to graph connectivity test. Our experiments demonstrate that for a variety of large graphs, the expected query counts required to establish graph connectivity are tiny and the query count distributions exhibit clear *long-tail* patterns. Essentially, the query limit is not bounding most of the time.

Lastly, the query limit is practically well motivated. In the context of our cyber-motivated model, the edge queries are answered by human efforts. Imposing a query limit is practically helpful to facilitate this algorithm-human collaboration. We prefer that the IT admin be able to specify the query limit before launching the interactive session (i.e., "my time allocation allows at most 10 queries").

Summary of Results

- We propose a combinatorial optimisation model called *Limited Query Graph Connectivity Test*, motivated by a cyber security use case on defending Active Directory managed Windows domain network.
- We show that query count minimization is #P-hard.
- We propose an empirically scalable exact algorithm. It can optimally solve practical-scale large graphs when the query limit is small. Even when we set the limit to infinity, our algorithm is capable of optimally solving several fairly large instances, including one with 18795 edges. (Recall that past works on exact algorithms only experimented on tiny instances with at most 20 edges/bits (Fu et al. 2017; Ben-dov 1981; Reinwald and Soland 1966; Breitbart and Reiter 1975).)
- Our exact algorithm iteratively generates more and more expensive policy that eventually converges to the optimal policy. When the exact algorithm is not scalable, its intermediate results serve as performance lower bounds.
- We propose three heuristics. Our best-performing heuristic is via imposing an artificially small query limit on the exact algorithm. The other two heuristics are based on reinforcement learning (RL) and Monte Carlo tree search (MCTS), where the action space is reduced with the help of query limit. We experiment on a wide range of practical graphs, including road and power networks, Python package dependency graphs and Microsoft Active Directory graphs. We conduct a comprehensive survey on existing heuristics and approximation algorithms on sequential testing, learning with attribute costs and stochastic Boolean function evaluation. Our heuristic building on the exact algorithm outperforms all, surpassing RL, MCTS and 8 heuristics ported from literature.
- Our techniques have the potential to be applicable to other models related to sequential testing, learning with attribute costs and stochastic Boolean function evaluation. The high-level idea of using query limit to derive empirically scalable exact algorithm is general. Using the query limit as the technical tool to derive high-quality heuristic and performance lower bound is also general, and it is applicable to settings without query limit.

Scalable Exact Algorithm

We preface our exact algorithm with a hardness result.

Theorem 1. *For the Limited Query Graph Connectivity Test problem, it is #P-hard to compute the minimum expected number of queries.*

(Fu et al. 2017) already derived a #P-hardness proof for a similar model. *The assumption of exponential costs is core to the authors' hardness proof, which is restrictive and not applicable for many practical applications.* Our proof works for unit cost, so our result is stronger.

Outer Loop: Identifying Paths/Cuts Worthy of Consideration and Exactness Guarantee

We first propose a systematic way for figuring out which paths/cuts are relevant to the optimal query policy. A path/-cut is relevant if it is referenced by the query policy – the policy terminates after establishing the path/cut. In other words, the policy tree's leaf nodes correspond to the path/cut. Generally speaking, when the query limit is small, only a few paths/cuts are relevant. As an extreme example, if the query limit is 2, then at most there are 4 leaf nodes in the optimal policy tree, which correspond to a total of 4 paths/cuts ever referenced. The challenge is to identify, continuing on the above example, which 4 are used among the *exponential* number of paths/cuts. Restricting attention to relevant paths/cuts will greatly reduce the search space, as we only need to focus on querying the edges that are part of them.

Key observation: Our terminating condition is to find a path, or find a cut, or reach the query limit. Finding a path can be reinterpreted as *disproving all cuts*. Similarly, finding a cut can be reinterpreted as *disproving all paths*.

Suppose we have a path set P and a cut set C, we introduce a *cheaper* (*cheaper or equal*, to be more accurate) problem, which is to come up with an optimal query policy that disproves all paths in P, or disproves all cuts in C, or reach the query limit. We use $\pi(P, C)$ to denote the optimal policy for this cheaper task, and use $c(P, C)$ to denote the minimum number of queries in expectation under $\pi(P, C)$. We use π^* to denote the optimal policy for the original problem, and use c^* to denote the minimum number of queries in expectation under π^*.

Proposition 1. *Given $P \subset P'$ and $C \subset C'$, we must have $c(P, C) \leq c(P', C')$, which also implies $c(P, C) \leq c^*$.*

Proof. The optimal policy $\pi(P', C')$ can still be applied to the instance with less paths/cuts to disprove. $\pi(P', C')$ has the potential to terminate even earlier when there are less to disprove. Lastly, c^* is just $c(\text{all paths}, \text{all cuts})$. □

Suppose we have access to a subroutine that calculates $\pi(P, C)$ and $c(P, C)$, which will be introduced soon in the remaining of this section. Proposition 1 leads to the following iterative exact algorithm. Our notation follow Figure 1.

1. We initialize an arbitrary path set P and an arbitrary cut set C. In our experiments, we simply start with singleton sets (one shortest path and one minimum cut).
2. Compute $\pi(P, C)$ and $c(P, C)$; $c(P, C)$ becomes the best lower bound on c^* so far.
3. Go over all the leaf nodes of the policy tree behind $\pi(P, C)$ that terminate due to successfully disproving either P or C. Suppose we are dealing with a node x that has disproved all paths in P. At node x, we have the results of several queries (i.e., when going from the policy tree's root node to node x, any left turn corresponds to an ON edge, and any right turn corresponds to an OFF edge). We check whether the confirmed OFF edges at node x form a cut. If so, then we have not just disproved all paths **in** P, but also disproved all paths **outside** P as well. If the confirmed OFF edges do not form a cut, then that means we can find a path that has yet been disproved. We add this path to P — our policy made a mistake (prematurely declared DONE) because we have not considered this newly identified path.

   We add new cuts to C using the same method.
4. If new paths/cuts were added, then go back to step 2 and repeat. If no new paths/cuts were added, then that means when $\pi(P, C)$ terminates, it has found a path, or found a cut, or reached the query limit. So $\pi(P, C)$ is a feasible query policy for the original problem. Proposition 1 shows that $c(P, C)$ never exceeds c^*, so $\pi(P, C)$ must be an optimal policy for the original problem.

Below we give an example illustrating that our proposed approach is able to successfully pick out only a small number of relevant paths/cuts. We apply our exact algorithm on a graph called EUROROAD (a road network with 1417 undirected edges (Kunegis 2022)). For $B = 10$, when the exact algorithm terminates, the final path set size is only 39 and the final cut set size is only 18.

Inner Loop: Designing Policy Tree Structure

We now describe the subroutine that computes $\pi(P, C)$ and $c(P, C)$ given the path set P and the cut set C as input.

This subroutine is further divided into two subroutines. First, we need to come up with an optimal *tree structure* (i.e., the shape of the policy tree). After that, we need to decide how to optimally place the queries into the tree structure.

We first focus on finding the optimal tree structure. Every policy tree is binary, but it is generally not going to be a complete tree of B layers (unless we hit the query limit 100% of the time). For example, for the optimal policy tree in Figure 1b, the left branch has only one node, while the right branch is deeper and more complex. The reason we care about the tree structure is that a small tree involves less decisions, therefore easier to design than a complete tree.

Given P and C, we define a policy tree to be *correct* if all the DONE nodes are labelled correctly (i.e., it is indeed done when the policy tree claims done). A correct policy tree does not have to reach conclusion in every leaf node (we allow a chain of queries to end up inconclusively). We also allow DONE nodes to have children, but they must all be DONE. We use the instance from Figure 1a as an example. Suppose P is the set of all paths and C is the set of all cuts. Figure 2a is a correct policy tree, because when it claims DONE, a is ON, which correctly disproves every cut in C. As you can

see, we allow the right branch to contain a partial policy that is inconclusive (after querying b, the remaining policy is not specified). Figure 2b is an incorrect policy tree because the DONE node on the right-hand side is a wrong claim. a being OFF is not enough to disprove every path in P.

(a) A correct policy tree (b) An incorrect policy tree

Figure 2: We consider the instance in Figure 1a. $P = \{(a), (b, c)\}$ and $C = \{(a, b), (a, c)\}$

Any policy tree for the original problem must be a correct policy tree. The cost of a correct policy tree has the following natural definition. DONE nodes have no costs. Every query node costs 1 and is weighted according to the probability of reaching the node. Given a tree structure S, we define $T(S, P, C)$ to be the minimum cost correct policy tree, as the result of optimally "filling in" queries and DONE into the tree structure S, subject to the only constraint that the resulting tree must be correct with respect to P and C. We define $c(S, P, C)$ to be the cost of $T(S, P, C)$.

Proposition 2. *For any tree structure S, path set P and cut set C, we have $c(S, P, C) \leq c(P, C)$.*

Proof. The policy tree T^* behind $c(P, C)$ can be imposed onto the tree structure S and results in a correct policy tree. For example, Figure 2a is a correct policy tree as a result of imposing Figure 1b to this new structure. Basically, any node position in S that is not in T^* can be filled in with DONE (since T^*'s leave must be DONE or has already reached the depth limit). Adding DONE does not incur any cost. Some query nodes from T^* may be dropped if their positions are not in S, which reduces cost. So T^* can be converted to a correct policy tree with structure S that is not more expensive. □

Proposition 3. *If S is a subtree of S', then $c(S, P, C) \leq c(S', P, C)$.*

Proof. $T(S', P, C)$ can be imposed onto S. The resulting tree is still correct and never more expensive. □

We will discuss how to calculate $T(S, P, C)$ and $c(S, P, C)$ (i.e., how to optimally fill in a given tree structure) toward the end of this section. Assuming access to $T(S, P, C)$ and $c(S, P, C)$, Proposition 2 and 3 combined lead to the following iterative algorithm for designing the optimal tree structure:

1. We initialize an arbitrary tree structure S. In experiments, we use a complete binary tree with 4 layers.
2. P and C are given by the algorithm's outer loop. We calculate $c(S, P, C)$, which becomes the best lower bound on $c(P, C)$ so far.
3. Go over all the leaf nodes of $T(S, P, C)$. If there is a leaf node that is not conclusive (not DONE and not reaching the query limit), then we expand this node by attaching two child nodes to the tree structure. That is, in the next iteration, we need to decide what queries to place into these two new slots. The cost for the next iteration never decreases according to Proposition 3.
4. If the tree was expanded, then go back to step 2 and repeat. If no new nodes were added, then that means $T(S, P, C)$ involves no partial decision. Its cost $c(S, P, C)$ must be at least $c(P, C)$ as $c(P, C)$ is supposed to be the minimum cost for disproving either P or C (or reach the query limit). This combined with Proposition 2 imply that $c(S, P, C) = c(P, C)$.

Earlier, we mentioned our algorithm's running details on a graph called EUROROAD with $B = 10$. For this instance, under our exact algorithm, the final tree structure size is 91. On the contrary, if we do not use the above "iterative tree growth" idea and simply work on a complete tree, then the number of slots is $2^{10} - 1 = 1023$. Our approach managed to significantly reduce the search space.

In the above algorithm description, we separated the "outer" and "inner" loops. This is purely for cleaner presentation. In implementation, essentially what we do is to iteratively generate the best policy tree so far given the current path set P, the current cut set C and the current tree structure S. If the resulting policy tree makes any wrong claims (claiming DONE prematurely), then we expand the path/cut set. If the tree goes into any inconclusive situation (reaching the leave position and still cannot claim DONE), then we expand the tree structure. During this iterative process, we keep getting equal or higher cost based on Proposition 1 and 3. When the process converges, which is *theoretically* guaranteed (*not practically*), we have the optimal query policy. The exact algorithm can be interrupted anytime and the intermediate results serve as performance lower bounds.

Optimal Correct Tree via Integer Program

We now describe the last subroutine. Given a path set P, a cut set C and a tree structure S, we aim to build the minimum cost correct policy tree $T(S, P, C)$.

Our technique is inspired by existing works on building optimal decision trees for machine learning classification tasks using integer programming (Bertsimas and Dunn 2017; Verwer and Zhang 2019). The works on optimal decision trees studied how to select the features to test at the tree nodes. The learning samples are routed down the tree based on the selected features and the feature test results. The objective is to maximize the overall accuracy at the leaf nodes. We are performing a very similar task. The objective is no longer about learning accuracy but on minimizing tree cost in the context of our model. Our model is as follows.

We first present the **variables**. Let E^R be the set of edges referenced in either P or C. For every tree node $i \in S$, for every edge $e \in E^R$, we define a binary variable $v_{e,i}$. If $v_{e,i}$ is 1, then it means we will perform query e at node i. We introduce another variable $v_{\text{DONE},i}$, which is 1 if and only if it is correct to claim DONE at node i.

Next, we present **constraints**.

Every node is either a query node or DONE, which implies $\forall i, \sum_{e \in E^R} v_{e,i} + v_{\text{DONE},i} = 1$.

If a node is DONE, then all its descendants should be DONE, which implies $\forall i, v_{\text{DONE},i} \geq v_{\text{DONE,PARENT}(i)}$.

Along the route from root to any leaf, each edge should be queried at most once. Let $\text{ROUTE}(i)$ be the set of nodes along the route from root to node i (inclusive). We have $\forall i \in \text{LEAVES}, \forall e \in E^R, \sum_{j \in \text{ROUTE}(i)} v_{e,j} \leq 1$.

DONE claims must be correct. Let LNODES be the set of tree nodes who are left children of their parents. If we claim DONE at node $i \in$ LNODES and $\text{PARENT}(i)$ is not already DONE, then we must have disproved all cuts in C once reaching i, since $\text{PARENT}(i)$'s query result is ON. To verify that we have disproved all cuts in C, we only need to consider the ON edges confirmed once reaching i. The queries that resulted in ON edges are queried at the following nodes $N(i) = \{\text{PARENT}(j) | j \in \text{LNODES} \cap \text{ROUTE}(i)\}$. Given a CUT (interpreted as a set of edges), to disprove it, all we need is that at least one edge in the cut has been queried and has returned an ON result, which is $\sum_{j \in N(i), e \in \text{CUT}} v_{e,j} \geq 1$.

We only need to verify that DONE is correctly claimed when it is claimed for the first time, which would be for nodes who satisfy that $v_{\text{DONE},i} - v_{\text{DONE,PARENT}(i)} = 1$. In combination, for node $i \in$ LNODES, we have the following constraint: $\forall \text{CUT} \in C, \sum_{j \in N(i), e \in \text{CUT}} v_{e,j} - (v_{\text{DONE},i} - v_{\text{DONE,PARENT}(i)}) \geq 0$. Note that $v_{\text{DONE},i} - v_{\text{DONE,PARENT}(i)}$ is 0 when we are not dealing with a node that claims DONE for the first time, which would disable the above constraint as it is automatically satisfied.

The above constraints only referenced cuts. We construct constraints in a similar manner for disproving paths.

The **objective** is to minimize the tree cost. The cost of node i is simply $\sum_{e \in E^R} v_{e,i}$. The probability of reaching a node only depends on the input tree structure so it is a constant. We use $\text{PROB}(i)$ to denote the probability of reaching node i. For example, if it takes 3 left turns (3 ON results) and 2 right turns (2 OFF results) to reach i, then $\text{PROB}(i) = p^3(1-p)^2$.

Heuristics

There are a long list of existing heuristics and approximation algorithms from literature on sequential testing, learning with attribute costs and stochastic Boolean function evaluation. (Ünlüyurt 2004)'s survey mentioned heuristics from (Jedrzejowicz 1983; Cox, Qiu, and Kuehner 1989; Sun, Chiu, and Cox 1996). Approximations algorithms were proposed in (Allen et al. 2017; Deshpande, Hellerstein, and Kletenik 2014; Kaplan, Kushilevitz, and Mansour 2005; Golovin and Krause 2011). We *select* and *port* 8 heuristics/approximation algorithms from literature for comparison. Below, we present one heuristic, referred to as H1. H1 is *impressively elegant* and experimentally it is the *dominantly better* existing heuristic. The definition is worded in the context of our graph model.

Definition 2 (H1 (Jedrzejowicz 1983), also independently discovered in (Cox, Qiu, and Kuehner 1989))**.** We find the path with the minimum number of hidden edges from s to t. The path is not allowed to contain OFF edges. We also find the cut with the minimum number of hidden edges between s to t. The cut is not allowed to contain ON edges. The path and the cut must intersect and the intersection must be a hidden edge. We query this edge.

Below we present three new heuristics for our model.

Heuristic based on the exact algorithm: We pretend that there are only B' queries left ($B' < B$) and apply the exact algorithm. B' needs to be small enough so that the exact algorithm is scalable. Once we have the optimal policy tree, we query the edge specified in the tree root, and then regenerate the optimal policy tree for the updated graph, still pretending that there are only B' queries left. We refer to this heuristic as TREE.

Reinforcement learning: (Yu et al. 2023) applied reinforcement learning to sequential medical tests involving 6 tests. We cannot directly apply reinforcement learning to our model, as we have a lot more than 6 queries to arrange. For our case, the action space is huge, as large graphs contain tens of thousands of edges. The key of our algorithm is a heuristic for limiting the action space. We use H1 to heuristically generate a small set of top priority edges to select from. Specifically, we generate H1's policy tree assuming a query limit of B'. All edges referenced form the action space. The action space size is at most $2^{B'} - 1$.

It is also not scalable to take the whole graph as *observation*. Fortunately, *the query limit comes to assist*. Our observation contains $B - 1$ segments as there are at most $B - 1$ past queries. Each segment contains $3 + (2^{B'} - 1)$ bits. 3 bits are one hot encoding of query result (not queried yet, ON, OFF). $2^{B'} - 1$ bits encode the action.

Since the action space always contains H1's default action, in experiments, when we apply Proximal Policy Optimization (PPO) (Schulman et al. 2017), the agent ended up cloning H1 and we cannot derive better/new heuristics. We instead apply Soft Actor-Critic for Discrete Action (Christodoulou 2019), as it involves entropy maximisation, which encourages the agent to deviate from H1. We managed to obtain better policy than H1 on many occasions.

Monte Carlo tree search: We use the same heuristic to limit the action space (same as RL). We apply standard epsilon-greedy Monte Carlo tree search.

Experiments

We experiment on a wide range of practical graphs, including road and power networks (Davis and Hu 2011; Kunegis 2022; Rossi and Ahmed 2015; Watts and Strogatz 1998; Dembart and Lewis 1981), Python package dependency graphs (pydeps 2022), Microsoft Active Directory attack graphs (DBCreator 2022; Carolo 2022; Guo et al. 2022; Goel et al. 2022; Guo et al. 2023; Goel et al. 2023) and more (Allard et al. 2019). For all experiments, we set $p = 0.5$. Sources and destinations are selected randomly. For Microsoft Active Directory attack graphs, the destination is set to be the admin node representing the highest privilege.

Our results are summarized in Table 1. *When $B = 5$, our exact algorithm scales for all 25 graphs.* The column OTHER shows the best performance among 8 heuristics from

	Size		$B = 5$		$B = 10$				
	n	m	Exact	Other	Exact/LB	Tree	RL	MCTS	Other
Road networks - Undirected									
USAIR97	333	2126	**3.813**	**3.813**	4.555	**4.986**	5.029	4.992	4.994
EUROROAD	1175	1417	**1.938**	**1.938**	**2.109**	**2.109**	**2.109**	**2.109**	**2.109**
MINNESOTA	2643	3303	**3.938**	**3.938**	4.785	5.506	5.525	5.506	**5.477**
Power networks - Undirected									
POWER	4942	6594	**3.625**	**3.625**	4.367	**4.615**	4.617	**4.615**	**4.615**
BCSPWR01	40	85	**3.125**	**3.125**	**3.541**	**3.541**	3.547	**3.541**	**3.541**
BCSPWR02	50	108	**2.625**	**2.625**	**3.178**	**3.178**	3.207	3.229	**3.178**
BCSPWR03	119	297	**3.938**	**3.938**	4.883	**5.816**	5.895	5.887	5.912
BCSPWR04	275	943	**3.938**	**3.938**	5.188	**6.074**	6.096	6.086	6.080
BCSPWR05	444	1033	**2.938**	**2.938**	3.555	4.551	4.543	4.561	**4.539**
BCSPWR06	1455	3377	**2.625**	**2.625**	3.164	**3.197**	3.211	3.246	3.201
BCSPWR07	1613	3718	**2.625**	2.689	3.346	**3.461**	3.500	3.564	3.625
BCSPWR08	1625	3837	**4.188**	**4.188**	4.945	**5.996**	6.174	6.266	6.414
BCSPWR09	1724	4117	**3.938**	**3.938**	4.664	**5.891**	5.895	5.924	**5.891**
BCSPWR10	5301	13571	**4.625**	**4.625**	5.270	7.428	7.393	**7.350**	7.553
Miscellaneous small graphs - Directed except for Chesapeake									
BUGFIX	13	28	**2.438**	**2.438**	**2.715**	**2.715**	**2.715**	**2.715**	**2.715**
FOOTBALL	36	118	**1.750**	**1.750**	**1.750**	**1.750**	**1.750**	**1.750**	**1.750**
CHESAPEAKE	40	170	**3.813**	**3.813**	4.594	**5.090**	5.152	5.199	5.113
CATTLE	29	217	**2.875**	**2.875**	3.721	**4.131**	4.166	4.158	4.188
Python dependency graphs - Directed									
NETWORKX	423	908	**2.000**	**2.000**	**2.078**	**2.078**	**2.078**	**2.078**	**2.078**
NUMPY	429	1208	**2.688**	**2.688**	3.418	**3.688**	3.717	3.709	3.719
MATPLOTLIB	282	1484	**2.188**	**2.188**	**2.520**	**2.520**	**2.520**	2.539	2.537
Active Directory graphs - Directed									
R2000	5997	18795	**1.938**	**1.938**	**1.938**	**1.938**	**1.938**	**1.938**	**1.938**
R4000	12001	45781	**2.688**	**2.688**	3.268	**3.344**	3.352	**3.344**	**3.344**
ADS5	1523	5359	**2.688**	**2.688**	3.287	**3.391**	3.402	**3.391**	**3.391**
ADS10	3015	12776	**3.375**	**3.375**	3.957	**4.123**	**4.123**	**4.123**	**4.123**

Table 1: Expected query count under different algorithms: EXACT=Exact algorithm; EXACT/LB=Either exact (bold) or lower bound; OTHER=Best out of 8 heuristics from literature (those underlined are **not** achieved by H1); TREE=Heuristic based on the exact algorithm; RL and MCTS are self-explanatory

literature. Besides H1 from Definition 2, the other 7 heuristics are from (Kaplan, Kushilevitz, and Mansour 2005; Allen et al. 2017; Deshpande, Hellerstein, and Kletenik 2014; Golovin and Krause 2011; Fu et al. 2017). For $B = 5$, H1 performs the best among all existing heuristics for every single graph. Actually, H1 produces the optimal results (confirmed by our exact algorithm) in all but one graph (BCSPWR07).

When $B = 10$, our exact algorithm produces the optimal policy for 8 out of 25 graphs (indicated by bold numbers in column EXACT/LB). For the remaining graphs, the exact algorithm only produces a lower bound (we set a time out of 72 hours). *The achieved lower bounds have high quality* (i.e., close to the achievable results).

When $B = 10$, we compare our best-performing heuristic TREE against the best existing heuristic (best among 8). TREE wins for 11 graphs, loses for 2 graphs, and ties for 12 graphs. It should be noted that in the above, we are comparing TREE against the **best** existing heuristic (best among 8). If we are comparing against any **individual** heuristic, then our "winning rate" would be even higher.

RL and MCTS perform worse than TREE. Nevertheless, if we take the better between RL and MCTS, the results outperform the existing heuristics (best among 8). We suspect that the "long-tail" pattern of the query costs causes difficulty for both RL and MCTS. The real advantage of RL and MCTS is that they are model-free. For example, model generalisation involving interdependent edges can easily be handled by RL and MCTS.

Lastly, our exact algorithm can find the optimal solution for BUGFIX, R2000, FOOTBALL *and* NETWORKX *even without query limit*. Among them, R2000 is the largest with 18795 edges.

To summarize, **our exact algorithm can handle significantly larger graphs** compared to experiments from existing literature using at most 20 edges (Ben-dov 1981; Reinwald and Soland 1966; Breitbart and Reiter 1975). Even when our exact algorithm does not terminate, it produces a **high-quality lower bound**. Our heuristics, especially TREE, **outperforms all existing heuristics** from literature.

Acknowledgments

Frank Neumann has been supported by the Australian Research Council (ARC) through grant FT200100536.

References

Allard, T.; Alvino, P.; Shing, L.; Wollaber, A.; and Yuen, J. 2019. A dataset to facilitate automated workflow analysis. *PloS one*, 14(2): e0211486.

Allen, S. R.; Hellerstein, L.; Kletenik, D.; and Ünlüyurt, T. 2017. Evaluation of Monotone DNF Formulas. *Algorithmica*, 77(3): 661–685.

Ben-dov, Y. 1981. A branch and bound algorithm for minimizing the expected cost of testing coherent systems. *European Journal of Operational Research*, 7(3): 284–289.

Bertsimas, D.; and Dunn, J. 2017. Optimal classification trees. *Machine Learning*, 106(7): 1039–1082.

Breitbart, Y.; and Reiter, A. 1975. A Branch-and-Bound Algorithm to Obtain an Optimal Evaluation Tree for Monotonic Boolean Functions. *Acta Inf.*, 4(4): 311–319.

Carolo, N. 2022. https://github.com/nicolas-carolo/adsimulator.

Christodoulou, P. 2019. Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207*.

Cox, L. A.; Qiu, Y.; and Kuehner, W. 1989. Heuristic least-cost computation of discrete classification functions with uncertain argument values. *Annals of Operations research*, 21(1): 1–29.

Davis, T. A.; and Hu, Y. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1).

DBCreator. 2022. https://github.com/BloodHoundAD/BloodHound-Tools/tree/master/DBCreator.

Dembart, B.; and Lewis, J. 1981. BCSPWR: Power network patterns. https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcspwr/bcspwr.html.

Deshpande, A.; Hellerstein, L.; and Kletenik, D. 2014. Approximation algorithms for stochastic boolean function evaluation and stochastic submodular set cover. In *Proceedings of the twenty-fifth annual ACM-SIAM Symposium on Discrete Algorithms*, 1453–1466. SIAM.

Fu, L.; Fu, X.; Xu, Z.; Peng, Q.; Wang, X.; and Lu, S. 2017. Determining source–destination connectivity in uncertain networks: Modeling and solutions. *IEEE/ACM Transactions on Networking*, 25(6): 3237–3252.

Goel, D.; Neumann, A.; Neumann, F.; Nguyen, H.; and Guo, M. 2023. Evolving Reinforcement Learning Environment to Minimize Learner's Achievable Reward: An Application on Hardening Active Directory Systems. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2023, Lisbon, Portugal, July 15-19, 2023*, 1348–1356. ACM.

Goel, D.; Ward-Graham, M. H.; Neumann, A.; Neumann, F.; Nguyen, H.; and Guo, M. 2022. Defending active directory by combining neural network based dynamic program and evolutionary diversity optimisation. In *GECCO '22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*, 1191–1199. ACM.

Golovin, D.; and Krause, A. 2011. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *Journal of Artificial Intelligence Research*, 42: 427–486.

Guo, M.; Li, J.; Neumann, A.; Neumann, F.; and Nguyen, H. 2022. Practical Fixed-Parameter Algorithms for Defending Active Directory Style Attack Graphs. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, 9360–9367. AAAI Press.

Guo, M.; Ward, M.; Neumann, A.; Neumann, F.; and Nguyen, H. 2023. Scalable Edge Blocking Algorithms for Defending Active Directory Style Attack Graphs. In *The 37th AAAI Conference on Artificial Intelligence (AAAI), Washington DC, USA*.

Jedrzejowicz, P. 1983. Minimizing the Average Cost of Testing Coherent Systems: Complexity and Approximate Algorithms. *IEEE Transactions on Reliability*, R-32(1): 66–70.

Kaplan, H.; Kushilevitz, E.; and Mansour, Y. 2005. Learning with attribute costs. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, 356–365.

Kunegis, J. 2022. http://konect.cc/networks.

pydeps. 2022. https://github.com/thebjorn/pydeps.

Reinwald, L. T.; and Soland, R. M. 1966. Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time. *J. ACM*, 13(3): 339–358.

Rossi, R. A.; and Ahmed, N. K. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms.

Short, M. W.; and Domagalski, J. E. 2013. Iron deficiency anemia: evaluation and management. *American family physician*, 87(2): 98–104.

Sun, X.; Chiu, S. Y.; and Cox, L. A. 1996. A hill-climbing approach for optimizing classification trees. In *Learning from Data*, 109–117. Springer.

Ünlüyurt, T. 2004. Sequential testing of complex systems: a review. *Discrete Applied Mathematics*, 142(1-3): 189–205.

Verwer, S.; and Zhang, Y. 2019. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 1625–1632.

Watts, D. J.; and Strogatz, S. H. 1998. Collective dynamics of small-world networks. *nature*, 393(6684): 440–442.

Yu, Z.; Li, Y.; Kim, J.; Huang, K.; Luo, Y.; and Wang, M. 2023. Deep Reinforcement Learning for Cost-Effective Medical Diagnosis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.