Active Graph Reachability Reduction for Network Security and Software Engineering

Alice X. Zheng
Microsoft Research
Redmond, WA
alicez@microsoft.com

John Dunagan
Microsoft
Redmond, WA
jdunagan@microsoft.com

Ashish Kapoor
Microsoft Research
Redmond, WA
akapoor@microsoft.com

Abstract

Motivated by applications from computer network security and software engineering, we study the problem of reducing reachability on a graph with unknown edge costs. When the costs are known, reachability reduction can be solved using a linear relaxation of sparsest cut. Problems arise, however, when edge costs are unknown. In this case, blindly applying sparsest cut with incorrect edge costs can result in suboptimal or infeasible solutions. Instead, we propose to solve the problem via edge classification using feedback on individual edges. We show that this approach outperforms competing approaches in accuracy and efficiency on our target applications.

1 Introduction

Graph reachability reduction aims to remove edges from a graph so that it becomes more difficult to reach the rest of the graph starting from any source node. This has natural applications in many fields of science and engineering. In this paper, we concentrate on two such applications in the field of computer science, namely, that of protecting a computer network from security attacks, and that of reducing software binary dependency in order to limit the scope of potential failure.

Graph reachability reduction can be understood as performing a partial graph cut. In the classic graph-cut setup, the goal is to find a set of edges such that removing them will render the graph disconnected. The sparsest cut variant aims to cut the set of edges with minimum cost, where each edge is weighted with the cost of cutting that edge. However, in practice, edge costs can often be difficult to quantify because they indicate un-parametrizable preferences. The problem is exacerbated in large graphs with many edges where it is impractical to specify the cost of each edge. In such cases, blindly applying graph cut algorithms without knowing the right edge costs can result in cuts that are infeasible or otherwise undesirable.

In this paper, we demonstrate how such problems can be solved through binary edge classification with active learning. We apply our method to our target applications and demonstrate that it is much more efficient than the alternative approach of iterating between learning edge cost and performing sparsest cut.

We start by introducing our applications of interest in Section 2, then define the problem and solution approaches in Section 3. We verify the accuracy and efficiency of our solution on a synthetic graph as well as the real-world applications in Section 4.

2 Applications to Network Security and Software Engineering

Consider this potential attack method on a computer network that employs an organization-wide authentication scheme [Dunagan *et al.*, 2009]:

1. Eve, the attacker, gains control of a single machine and lies in wait.
2. Unsuspecting user Alice logs on to the compromised machine and exposes her identity key to the attacker.
3. Eve impersonates Alice using her identity, thereby gaining access to other machines on the network on which Alice has administrative privilege.
4. Eve repeats the process starting from the newly compromised machines.

In this way, an initial compromise can snowball into a much larger compromise—an *identity snowball attack*.

To protect the network from these attacks, one may wish to modify existing security policies so that an initial attack would not spread to too many other machines [Dunagan *et al.*, 2009], but in a way such that the network still remains connected. This can be modeled as a graph reachability reduction problem. The relevant entities in an identity snowball attack can be captured in an *attack graph* ([Sheyner *et al.*, 2002; Ou *et al.*, 2005]) where nodes represent network entities (machines, users, or user groups) and edges represent interactions such as a user logging onto a machine or belonging to a group, or a user or group having administrative privileges on a machine. The direction of an edge indicates control, e.g., a log-in action gives the machine control over the user's identity. (See Figure 1 for an example.) Reachability in an attack graph implies potential compromise: if machine B is reachable from machine A through a sequence of login and admin edges, then machine B may be compromised if machine A

is compromised. Hence, reducing reachability on the attack graph limits the scope of such attacks.

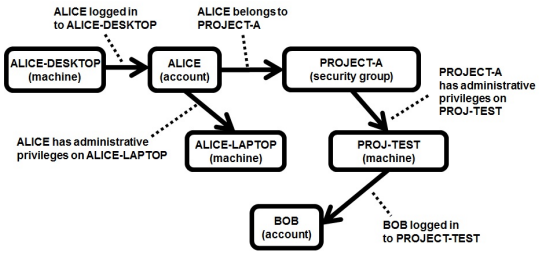

Figure 1: *An example attack graph on a computer network. Nodes represent entities on the network (machines, users, user groups) and edges represent interactions. The direction of an edge indicates control, e.g., a log-in action gives the machine control over the user's identity.*

However, it is difficult to specify the edge costs in this setting because they represent the implementability of security policy changes. For example, it might be easy to remove an edge that grants administrative privilege on a machine to all members of a large user group, because not all members of the group in fact need that privilege. But it would not be okay to bar a user from logging onto her own desktop machine. These decisions often need to be made on an edge-by-edge basis, and thus it is very difficult to specify cost for all the edges in a large attack graph.

Fortunately for the learning algorithm, the users of this algorithm—the network experts who are in charge of protecting the network—can provide a limited amount of feedback on a few edges. When presented with a small group of edges, the expert may be able to choose a few of them as "Ok to cut" and the rest "keep for now." The challenge remains how to find good edges to cut without knowing all of the edge costs in the graph.

As another example of graph reachability reduction, consider the software engineering task of reducing binary module dependencies. Large software systems are often divided into modules, each containing many classes or functions. A dependency is created when a function in module A calls a function in module B. Such dependencies between binary modules are necessary and useful, just like connections between networked computers and users are necessary. But binary dependencies can also raise reliability concerns: If the callee function fails and the error is not handled well, the caller could also fail. For this reason, it is often desirable to reduce the number of dependencies [Nagappan and Ball, 2007].

Although dependencies are easy to identify, determining the cost of removing a dependency is expensive, as it involves estimating the effort required in modifying the code. This can be done on a case-by-case basis for each dependency, but is difficult to do for all dependencies. Hence, it has traditionally taken significant manual work to determine the set of dependencies to remove. In this paper, we cast this as another instance of reachability reduction on a software dependency graph with unknown edge costs.

3 Solution Approaches

3.1 Background on Graph Algorithms

Intuitively, reducing reachability corresponds to making it harder to get from one starting point in a graph to another. There have been many formalizations of the notion of the reachability of a graph, many of which have been shown to be related to within a constant factor [Arora *et al.*, 2004]. One possible definition of graph reachability is the maximum concurrent flow [Shahrokhi and Matula, 1990] achievable within a graph. With this definition, reachability reduction corresponds to removing edges such that this maximum achievable flow is decreased. It is well-known that the set of edges that limits the maximum concurrent flow is also the set that yields the sparsest cut (or balanced cut). More formally, the dual LP to maximum concurrent flow is the sparsest cut LP [Shahrokhi and Matula, 1990].

Consider a graph $G = (V, E)$, where V denotes the set of vertices and E the set of directed edges. Let $c(e)$ denote the cost of cutting edge e. Assume there exists a set of *demands* between a set of source nodes and a set of target nodes in the graph. For example, in modeling the network security application, we can specify a unit demand between every pair of machine nodes, and in the binary dependency reduction application, there is demand between each pairs of binaries. The goal of sparsest cut is to determine the smallest set of edges whose removal would cut all non-zero demand. Here is one possible formulation of the sparsest cut LP:

$$\min_{d(e)} \quad \sum_{e \in E} c(e)d(e) \tag{1}$$

$$\text{subject to} \quad \sum_{e \in E} b(e)d(e) = 1 \tag{2}$$

$$d(e) \geq 0. \tag{3}$$

Here, $d(e)$ represents an edge distance assignment, where a greater distance denotes a more favorable edge to cut. The quantity $b(e)$ is equal to the number of shortest paths (between node pairs with non-zero demand) that pass through e; it can be interpreted as the "benefit" of cutting edge e.[1] Figure 2 essentially encodes the sparseness constraint of cutting edges (i.e., setting $d(e)$): under a fixed budget of 1 for the weighted sum of distances, it incentivizes the optimization procedure to set $d(e)$ higher for more central edges along the collection of shortest paths between nodes with non-zero demand.

3.2 Previous Approaches: Alternate

The above formulation of graph cut assumes all edge costs are known. When they are not known, one natural approach is to alternate between learning the edge costs (based on user feedback) and performing graph cut. Dunagan *et al.* [2009] describes the following algorithm called Heat-ray for preventing identity snowball attacks. We denote this algorithm as Alternate-HR. The algorithm first initializes the edge costs $c(e)$ to be uniform. Then, each iteration first runs sparsest cut

[1] In this formulation of sparsest cut, $b(e)$ depends on path lengths which are defined using $d(e)$. Hence the formulation as presented here is not an LP, but it does have an equivalent LP relaxation form. We choose to present this formulation for its readability.

Algorithm 1: Alternate-Heat-ray (Alternate-HR)

Input : Graph $G = (V, E), \alpha$.
Output: Graph $G' = (V, E')$.

Initialize $G' \leftarrow G$ costs $c(e) \leftarrow 1$ for $e \in E$.
repeat
  Stage 1:
    $d(e), b(e) \leftarrow$ `SparsestCutLP` $(G, c(\cdot))$
    Rank edges by descending $\max(b(e), -b(e)c(e))$
  Stage 2:
    $R \leftarrow$ user feedback on edges
    $c(e) \leftarrow$ `UpdateCost` (R) using Eqn (6)
until `Reachability` *(G')* $\leq \alpha$ *or all edges queried*

(cf. Eqn (1)–(3)) to rank the edges. This ranked list is presented to the user, who selectively labels a subset R of the top-ranked edges as "cut" or "keep." The feedback is then incorporated into a second set of linear constraints to solve for new estimates of edge cost.

$$b(e) - c(e) \leq -1 \quad \text{if } e \text{ is to be kept,} \tag{4}$$
$$b(e) - c(e) \geq 1 \quad \text{if } e \text{ is to be cut.} \tag{5}$$

Each edge is represented by a vector of edge features $\mathbf{x}_e$. The cost function is modeled as a linear function of the features: $c(e) = \mathbf{w}^T \mathbf{x}_e$. Let y_e denote the feedback on edge e: $y_e = 1$ if e is to be cut, and $y_e = -1$ if it is to be kept. Alternate-HR finds the $\mathbf{w}$ that minimizes the hinge loss of unsatisfied constraints, regularized by the norm of the weight vector (akin to linear SVMs):

$$\min_{\mathbf{w}} \sum_{e \in R} \max(0, 1 - y_e(b(e) - w^T \mathbf{x}_e)) + \lambda \mathbf{w}^T \mathbf{w}. \tag{6}$$

After learning the new cost function $c(e)$, the updated costs are used in the next round of sparsest cut optimization.

We can also formulate a much simpler version of the alternate approach. We call the algorithm Alternate-SimpleUpdate (Alternate-SU). It initializes all edges to have uniform cost μ. In the first stage, it computes the LP relaxation of sparsest cut and ranks the edges according to $d(e)$. The top edges are presented to the user for feedback. The costs of "cut" edges are then updated to γ, and those of "keep" edges are updated to κ, where $\gamma \leq \mu \leq \kappa$. Alternate-SU then recomputes $d(e)$ and repeats.

3.3 New Approach: ARRG

So far, our problem statement contains an interesting mixture of two disparate tasks: performing sparsest cut on a graph and learning the edge cost. Conceptually, graph reachability reduction is the end goal. However, the application requires that all edge cut decisions be feasible. In a graph cut setting, feasibility is modeled by the (unknown) cost of cutting an edge. Previous approaches solves the problem by folding the task of edge cost learning into an outer loop of sparsest cut. We now propose to instead fold the problem of graph reachability reduction into the process of learning the user's edge cut preference function, which may depend on its utility in the graph cut setting, as well as a number of other features.

Algorithm 2: Alternate-SimpleUpdate (Alternate-SU)

Input : Graph $G = (V, E), \mu, \gamma, \kappa, \alpha$.
Output: Graph $G' = (V, E')$.

Initialize $G' \leftarrow G$ and $c(e) \leftarrow \mu$ for $e \in E$.
repeat
  Stage 1:
    $d(e) \leftarrow$ `SparsestCutLP` $(G, c(\cdot))$
    Rank edges by descending $d(e)$
  Stage 2:
    $R \leftarrow$ user feedback on edges
    foreach *edge* e *in* R **do**
      if $R(e) =$ *Cut* **then** $c(e) \leftarrow \gamma$, $E' \leftarrow E' \backslash e$
      else if $R(e) =$ *Keep* **then** $c(e) \leftarrow \kappa$
until `Reachability` *(G')* $\leq \alpha$ *or* $E' = \emptyset$

In this case, it is likely the user herself does not have perfect knowledge of which edges should be cut. While the question of a noisy oracle is interesting from a theoretical perspective, in this paper we use active learning as a practical solution to the problem of helping an imperfect oracle in making better decisions, and, at the same time, efficiently and accurately learning from such oracles. As our experiments show, ARRG is much quicker at solving the problem than Alternate, because it makes more direct use of available features and feedback.

Let $D_c : E \rightarrow \{0, 1\}$ be the indicator function for the optimal cut-set under $c(e)$, i.e., $D_c(e) = 1$ if and only if e should be removed. Sparsest cut tries to obtain the entire D_c through global optimization. We propose to approximate $D_c(e)$ edge-by-edge with a classifier $f(\mathbf{x}_e)$, where $\mathbf{x}_e$ is a set of features for edge e. We train the classifier using active learning, where edge labels are provided by the user's feedback.

We call this method Active Reachability Reduction on Graphs (ARRG). In this paper, we investigate ARRG using active learning techniques for the Gaussian process classifier (GPC) and support vector machine (SVM). Let $\mathbf{X}_L = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, denote a set of labeled training data and $\mathbf{X}_U = \{\mathbf{x}_{n+1}, \ldots, \mathbf{x}_{n+m}\}$ a set of unlabled data for which the oracle may provide feedback. Active learning aims to select the unlabeled sample whose labeling would most improve the classifier. Numerous active learning criteria have been proposed for various classifiers. In our experiments, we use the margin (distance to the closest point) for SVM and the entropy for GPC.

$$\text{SVM:} \qquad \mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathbf{X}_U} |f_{SVM}(\mathbf{x})| \tag{7}$$
$$\text{GPC:} \qquad \mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathbf{X}_U} H(q_{GPC}(\mathbf{x})) \tag{8}$$

Here, $H(\cdot)$ denotes entropy, $|f_{SVM}(\mathbf{x})|$ denotes the SVM margin, and $q_{GPC}(\mathbf{x})$ denotes the posterior distribution over the label of $\mathbf{x}$ inferred by Gaussian process classification (GPC).

ARRG-SVM and ARRG-GPC each has strengths and drawbacks. While the active learning criterion for SVM is simple to compute, the GPC criterion can provide better performance. This is because GPC provides the full posterior

Algorithm 3: ARRG

Input : Edge feature list $E = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\}$.
Output: Classifier $f(\mathbf{x})$.

Initialize training set $T \leftarrow \{(\mathbf{x}_{e_1}, 1), (\mathbf{x}_{e_2}, -1)\}$ with one "cut" edge and one "keep" edge.
repeat
  $f(\cdot) \leftarrow$ `TrainClassifier` (T)
  $e \leftarrow$ `ActiveLearningTopEdge` (f, E)
  $y_e \leftarrow$ user feedback on e
  $T \leftarrow T \cup \{(\mathbf{x}_e, y_e)\}$
until *convergence*

predictive distribution over the labels, thereby taking into account both the distance from the boundary as well as the variance around the estimate. Another notable difference is that SVM has the added complexity of choosing the kernel hyperparameters and the regularization parameter. Gaussian Processes on the other hand can simply use marginal likelihood to determine additional parameters (if any).

Like many classification approaches, the performance of ARRG depends on the quality of features. When available features correlate well with the feasibility of edge cuts, the user oracle can make better decisions, and ARRG is able to make better predictions. In our experiments, we use graph-based as well as non-graph-based features. Graph-based features measure the local or global role of the edge in the graph. Examples include (unweighted) in- and out-degrees of u and v, and the number of shortest paths e lies on (or $b(e)$ from Eqn (2)). Non-graph-based features can be attributes for the edge or its two end nodes. They represent any additional knowledge we have about entities in the graph that may have bearings on the edge cost. In the identity snowball attack prevention example, we include features such as edge type (login, admin, or group membership), and the function of the machine (normal user machine or server).

4 Experiments

We compare ARRG against Alternate and two other baselines on one synthetic dataset and our target applications in network security and software dependency reduction. The Marginal baseline is the classification accuracy of the simple marginal classifier that predicts all labels to be the more popular label. In all of our experiments, most of the edges are to be kept, and hence this baseline sets a high bar for ARRG to beat. We also compare against the performance of sparsest cut with uniform cost, which represents the baseline of ignoring edge costs.

Previewing our results, we find that ARRG outperforms its competitors both in efficiency (in terms of the amount of edge feedback needed) and in accuracy (which edges should be removed).

For the Alternate algorithms, we use the implementation of the sparsest cut LP as detailed in Dunagan *et al.* [2009], which modifies the objective to eliminate one of the constraints and transforms $d(e)$ to ensure non-negativity. We experimentally found that simple batch gradient descent converges to a usable solution within 10 steps with a step size of 0.01. For Alternate-SU, we rank the edges according to $d(e)$ and mark the top k edges as needing to be cut. Here k is taken to be ground truth number of edges that are labeled as "cut," giving sparsest cut an additional advantage over ARRG-GPC, which has no knowledge of the correct number of edges to cut. Note that this is equivalent to comparing ARRG against (close to) optimal sparsest cut solutions with varying strategies for learning edge cost.

For experiments with ARRG, we use the Gaussian process implementation of Rasmussen and Williams [2007] and the support vector machine implementation in libsvm [Chang and Lin, 2009]. Both classifiers are initialized with two randomly chosen examples, one from the positive (cut edge) class and one from the negative (keep edge) class. In our implementation, the mean and variance parameters in GPC are approximated using expectation propagation [Minka, 2001]. The regularization parameter C for SVM is tuned using leave-one-out cross-validation over a grid of values. All reported ARRG results are averaged over 20 runs with different random initializations. To improve readability, we omit plotting the standard errors, which in most settings are very tight. All experiments are conducted on the same sets of training and test data.

All experiments are conducted on a 3.0 Ghz dual Pentium Xeon machine with 8 GB of RAM running a 64 bit Windows Vista operating system and Matlab. Running time ranged from under a minute on the small synthetic graph to 30 minutes (for 300 edge queries) on the large identity snowball attack graph.

4.1 Experiment on a Synthetic Graph

First, we perform a sanity check using a simple synthetic graph of three clusters with roughly equal number of connections between clusters 1 and 2, and between 2 and 3. Each node has a position in the 2-D plane as well as a color. Edge cost (unknown to the algorithms) depend on the colors of the nodes. Clusters 2 and 3 have the same type and cluster 1 has another type. Hence the right cut is between clusters 1 and 2.

The edge features used in ARRG for edge $e = (u, v)$ are the in- and out-degrees of u and v, the number of all-pairs shortest paths that e lies on, the x- and y-positions of u and v, and the colors of u and v.

We first compare the performance of ARRG-SVM against ARRG-GPC, both with a linear kernel. We also look at the performance of GPC and SVM when edge queries are selected randomly (Random GPC and Random SVM). We find that both ARRG-GPC and ARRG-SVM reach perfect prediction accuracy, but ARRG-GPC enjoys two advantages over ARRG-SVM. First, ARRG-GPC requires slightly fewer edge labels. Second, ARRG-SVM performance fluctuates widely with few edge samples. This is due to the fact that when the number of labeled examples is low, cross-validation has difficulty determining C, the regularization parameter in SVMs. Consequently, we observe high variation in performance in the initial rounds of active learning. Performance of Random GPC and Random SVM are noticeably worse than their active learning counterparts. Overall all four algorithms quickly

pass the marginal baseline, and we find that ARRG-GPC provides the best combination of efficiency and accuracy.

Next, we compare ARRG-GPC against multiple iterations of Alternate-SU, which on each iteration receives information about the edge chosen by ARRG-GPC. Since the true edge cost is known for the synthetic graph, it is revealed to Alternate-SU. As shown in Figure 2, Alternate-SU takes much longer to find the correct cut than ARRG-GPC, even with the added benefit of knowing the true cost of the edge queried by ARRG-GPC as well as the total number of edges that should be cut. Alternate-SU requires around 100 edges (over 60% of edges in the graph) to reach 100% accuracy while ARRG-GPC requires only 13 edges. ARRG-GPC does suffer a bit with fewer than 5 labeled edges; it has lower accuracy than sparsest cut with uniform cost.

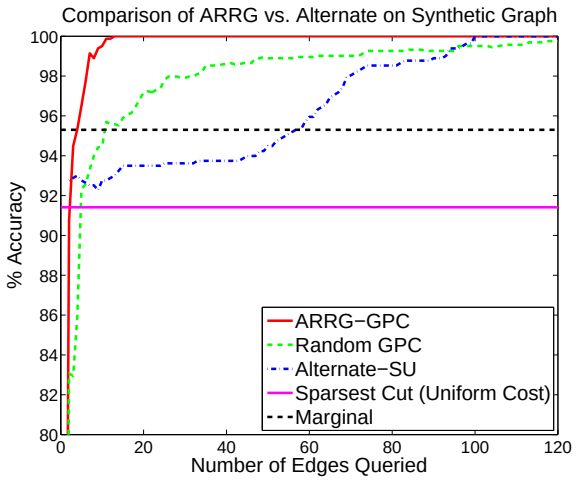

Figure 2: *Comparison of classification accuracy of ARRG-GPC and Alternate-SC.*

4.2 Reducing Software Module Dependencies

In this experiment, we study a set of modules in a large software system which has undergone continuous development over several years. As part of the development cycle, an effort was made to significantly reduce dependencies. We take two snapshots of the software, one before the dependency reduction effort and one after. The nodes in the graph are a portion of the modules that appear in both snapshots. If an edge from module A to module B is present in the before-shot but not the after-shot, that means module A no longer depends on module B in the new version of the software. All the algorithms we evaluate operate on edges present in the before-shot and try to predict which ones are no longer present in the after-shot. There are 400 nodes and 1004 edges, 287 of which are removed. Edges features are the in- and out-degrees of each module, the number of all-pairs shortest paths containing the edge, the frequency of calls from one module to the other, and cyclomatic complexity of the two modules. Cyclomatic complexity is a measure of source code complexity [T. McCabe, 1976]. Prior research has suggested that higher cyclomatic complexity leads to a higher likelihood of module failure; hence it becomes less desirable for other modules to depend on it.

Experimental results on the software module dependency graph are presented in Figure 3(a). The marginal baseline is 71.4%, whereas sparsest cut with uniform cost achieves 60.6% accuracy. As in the synthetic experiment, the performance of Alternate-SU steadily improves with more edge queries. but it does not exceed the marginal baseline until well after 200 edge queries. As before, ARRG-GPC starts out with low accuracy when very few edges are labeled, quickly exceeds Alternate-SU within 10 queries, and exceeds both the marginal baseline and Random GPC within 100 queries. The relative ranking of the various algorithms are roughly the same as before. ARRG-GPC is again the overall winner, but overall prediction accuracy is worse. This is because the classification task is much harder. The decision to remove certain software module dependencies was often based on considerations other than source code complexity. Hence the available features do not sufficiently represent the classification criteria. This is evident during training: the two classes of edges are not linearly separable, necessitating the use of a polynomial kernel of degree 2 in GPC as opposed to the linear kernel employed for the two previous experiments. This makes for a more difficult training process, which explains why ARRG-GPC requires more edge queries to exceed the baseline in this experiment. However, it is reasonable to expect overall accuracy to improve with more helpful features such as the failure rate of the software modules. Overall, the performance shows that ARRG is much more efficient than previous approaches at incorporating user feedback; it can provide better prediction results much sooner than competing algorithms.

4.3 Preventing Identity Snowball Attacks

In this experiment, we study an identity snowball attack graph based on data collected at a large organization. The graph contains login actions and administrative privileges recorded over the period of eight days. The full graph contains over 350K nodes of type machine, user, and group. The network activities constitute over 4.5 million edges. Table 1 lists detailed statistics.

For our first experiment, we take a subgraph of the full graph and compare ARRG-GPC against Alternate-SU as well as our simple baselines. This subgraph contains 3794 nodes (3037 machine nodes, 757 user nodes, and no group nodes) and 5597 edges (1291 login edges, 4306 user admin edges, and no group membership or group admin edges). A network security expert labeled 126 edges as being desirable to remove. As features for classification, an edge (u, v) is represented by the in- and out-degrees of u and v, the number of machine-to-machine shortest paths passing through the edge, the types of u and v (machine or user node), whether the machine acts as a server machine, and the number of times the user logs onto the machine (if it's a login edge).

Figure 3(b) plots results from running ARRG-GPC, Random GPC, and Alternate-SU on the identity snowball attack graph. As before, the y-axis denotes classification accuracy percentage over the entire graph and the x-axis denotes the number of edges queried. The marginal baseline of the majority label predictor is 97.7%. Sparsest cut with

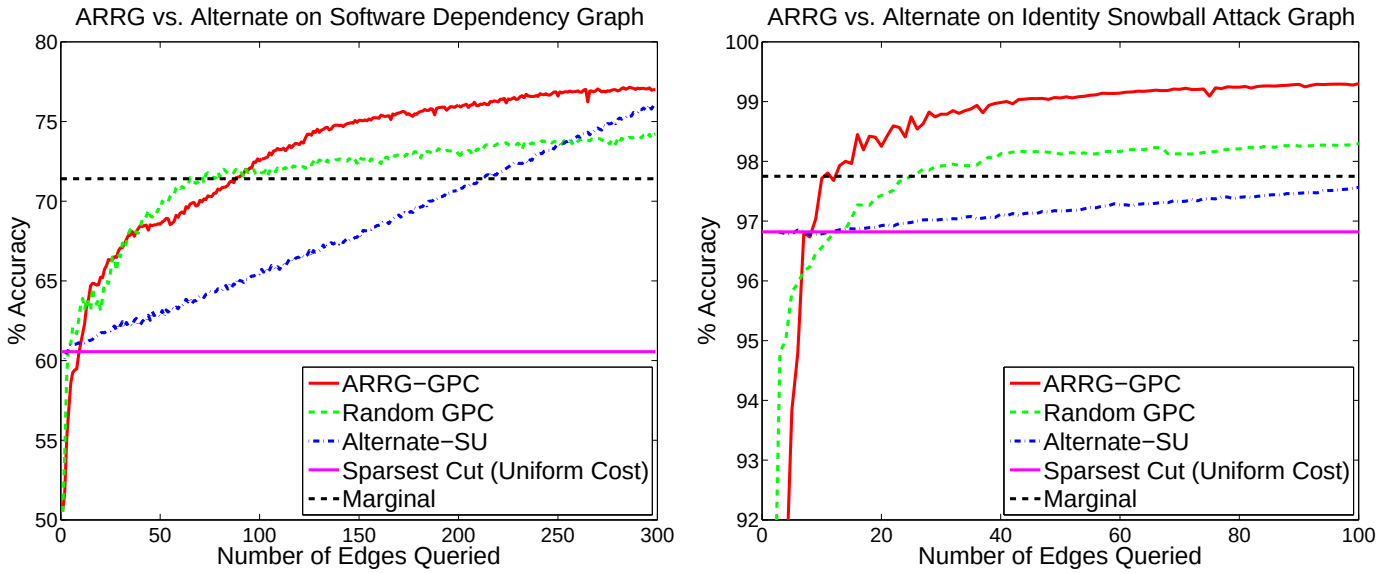

Figure 3: *(a) Edge removal accuracy on the software dependency graph. (b) Edge removal accuracy on a subgraph of the identity snowball attack graph.*

Machines	197,631
Accounts	91,563
Security Groups	62,725
Total Nodes	351,919
Unique Logins	130,796
AccountIsAdminOnMachine	309,182
SecurityGroupIsAdminOnMachine	380,320
AccountIsMemberOfSecurityGroup	3,695,878
Total Edges	4,516,176

Table 1: *Statistics of the identity snowball attack graph.*

uniform cost achieves 96.8% accuracy. As Alternate-SU receives more help with setting edge costs based on the ARRG-GPC edge selections[2], its classification accuracy improves steadily, but it never exceeds the marginal baseline within 100 edge queries. ARRG-GPC, on the other hand, starts out with very low accuracy when only two edges are labeled (59.2%), but quickly improves as more edges are labeled. Its average performance exceeds sparsest cut and Alternate-SU within 9 edge queries, and exceeds the marginal baseline at 13 queries. ARRG-GPC reaches an average accuracy of 99% within 45 queries. Random GPC fares relatively poorly; its average exceeds the sparsest cut uniform cost baseline at 12 edge queries and marginal baseline at 25 edge queries, but it never reaches 99% within 100 queries. All results are significant based on standard errors computed from 20 runs of each algorithm (starting with two randomly selected labeled edges each time).

In our second experiment with the identity snowball attack graph, we compare ARRG-GPC against Alternate-HR. The goal of this experiment is to study their relative efficiency, measured as the number of edge queries needed to reach a certain level of accuracy. In order to facilitate feedback gathering from the network security expert, the edges are generalized into edge groups – all the outgoing edges or all the incoming edges of a node. The network security expert is asked to supply one label for each edge group. The full graph is labeled based on feedback obtained in 10 iterations of Alternate-HR, where the expert was presented with a total of 6000 incoming and outgoing edge groups. Based on his feedback, 24804 edge groups were determined to be ok to remove and 184698 edge groups to be kept. Edge features include the number of shortest paths going through the edge, the type of the edge (login, group/user admin, or group

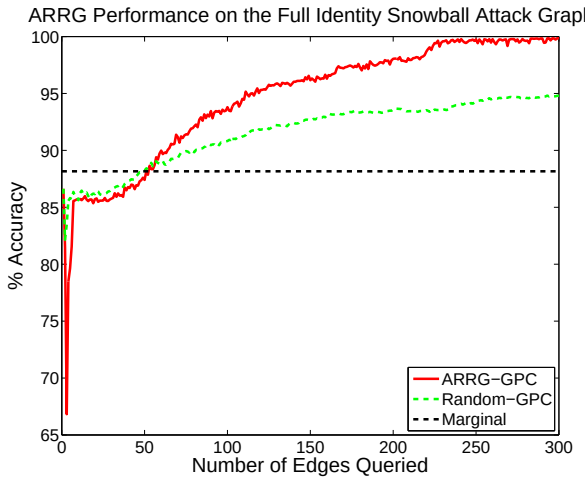

Figure 4: *ARRG performance on the identity snowball attack graph.*

[2]In the synthetic experiment, we know the true cost of all edges. In the two real applications, however, we do not have information about the true cost; instead, Alternate-SU sets the edge cost to be a small fixed number (0.01) if the oracle says "cut this edge" and a large fixed number (100) if the oracle says "keep this edge."

True Labels	Predicted Labels Keep	Cut
Keep	184,320	222
Cut	80	24578

Table 2: *Average confusion matrix of ARRG-GPC on the full identity snowball attack graph test set.*

membership), the in- and out-degrees of the node counted by edge type, and a binary indicator for outgoing edge group or incoming edge group. ARRG-GPC was trained using a polynomial kernel of degree 2. The sparsest cut stage of Alternate-HR is run with demand between only pairs of machine nodes. Due to the large size of the group, Alternate-HR computes $b(e)$ using shortest paths sampled from 100 randomly selected starting nodes.

The results are shown in Figure 4. After querying only 300 edges, ARRG-GPC is able to reach an average edge classification accuracy of 99.9%. In other words, after labeling on 4% of the number of edges required by Alternate-HR, our algorithm is able to provide edge cuts with close to 100% accuracy. Table 2 lists the confusion matrix averaged over 20 runs. To understand the high accuracy achieved by ARRG-GPC, we talked to the network security expert who labeled the edges and found that he looked for users who log on to many machines and removed all login privileges except for those to essential servers and the machines to which they log on most frequently. He also removed admin privileges for groups of large size. These criteria are well captured by the edge features.

5 Conclusion

In this paper we study applications of graph reachability reduction in network security and software engineering. We formulate the problem with unknown edge costs and propose a solution based on binary classificaiton with active learning (ARRG). We show that, on two real-world applications, ARRG clearly wins over competing approaches at the task of identifying good edges to cut in a graph with unknown (and non-uniform) edge costs. Our approach may seem intuitive in hindsight. However, all prior work in related areas have instead focused on the strategy of alternating between learning the metric and solving the main objective (graph cut in this case). Our success highlights the importance of aligning the learning algorithm with the type of available feedback.

References

S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. In *Symposium on Theory of Computing (STOC)*, 2004.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM—A Library for Support Vector Machines, Nov 2009.

John Dunagan, Alice X. Zheng, and Daniel R. Simon. Heatray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. In *Symposium on Operating Sytems Principles (SOSP)*, 2009.

T. Minka. Expectation Propagation for approximate Bayesian inference. In *Uncertainty in Artificial Intelligence (UAI)*, 2001.

N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007.

X. Ou, S. Govindavajhala, and A.W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *USENIX Security*, 2005.

Carl Edward Rasmussen and Christopher K. I. Williams. Documentation for GPML Matlab Code, June 2007.

F. Shahrokhi and D. Matula. The Maximum Concurrent Flow Problem. *Journal of the ACM (JACM)*, 37(2):318–334, 1990.

O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. Automated Generation and Analysis of Attack Graphs. In *IEEE Security and Privacy (IEEE Oakland)*, 2002.

T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.