The Automated Design of Network Graph Algorithms with Applications in Cybersecurity

by

Aaron Scott Pope

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 2, 2020

Keywords: Hyper-heuristics, Genetic Programming, Graph Algorithms, Network Security, Evolutionary Computation, Computer Security

Approved by

Dr. Daniel Tauritz, Chair, Associate Professor of Computer Science and Software Engineering
Dr. David Umphress, COLSA Corporation Cyber Security and Information Assurance Professor
Dr. Alice Smith, Joe W. Forehand/Accenture Distinguished Professor of Industrial and Systems Engineering
Dr. Gerry Dozier, Charles D. McCrary Eminent Chair Professor of Computer Science and Software Engineering
Dr. Alexander Kent, Director of Cybersecurity Engineering, Cardiac Rhythm and Heart Failure Division, Medtronic

Abstract

Graph representations and graph algorithms are commonplace in a wide variety of research domains, including computer and network security. Many problems can be expressed in terms of graphs in order to leverage the strengths of existing graph-based heuristics. Often, these applications employ general purpose, off-the-shelf graph algorithms that are agnostic of the particular problem domain. Customized heuristics can be developed that achieve improved performance by utilizing problem-specific knowledge, but this process can be expensive and time consuming. Hyper-heuristics can be used to automate this process to develop novel graph-based algorithms that are tailored to the specific application. This dissertation describes a number of contributions to the domains of evolutionary computation, hyper-heuristics, and cybersecurity. An evolutionary algorithm is combined with a graph partitioning approach to prescribe network access control configuration changes that reduce vulnerability to adversarial traversal while minimizing impact on legitimate users. A hyper-heuristic framework is detailed that automates the design and optimization of tailored graph algorithms and the potential of this framework is demonstrated on multiple network security applications. Graph generating algorithms are tailored to accurately model complex network behavior, both static and dynamic. Novel security metrics are produced that analyze network vulnerability to specific attack models. Graph partitioning heuristics are customized to reduce the application cost of network segmentation methods. Link prediction heuristics are automatically tailored for computer network anomaly detection applications. This work also contributes novel methods of improving hyper-heuristic performance on complex real-world applications by dynamically controlling the granularity of the heuristic search. Dynamic granularity control has the potential to improve the applicability and scalability of hyper-heuristic methods to a wide variety of application domains.

Acknowledgments

Portions of the work contained in this document were supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontracts 259565 and 570204, as well as the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project numbers 20180607ECR and 20170683ER.

I would like to thank my advisor, Dr. Daniel Tauritz, the rest of my committee, Dr. David Umphress, Dr. Alice Smith, Dr. Gerry Dozier, and Dr. Alexander Kent, and my University Reader, Dr. Roy Hartfield. I would also like to thank my collaborators at Los Alamos National Laboratory, Dr. Melissa Turcotte and Chris Rawlings. I am also grateful for the friends who encouraged me to persevere along the way including Anusha Sankara, Alex Bertels, Calvin Ardi, Alex Bolton, and countless others. I would also like to thank my fiancée, Charlotte, for her patience and understanding. Finally, I would like to dedicate this dissertation to my children, Annaliese Raine and Grayson Scott, who made all of the effort worthwhile.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are a powerful and widely used representation in many scientific research domains. Examples of concepts that are commonly represented as graphs include social networks, power grids, and transportation systems. A graph representation provides an abstraction that makes these concepts easier to understand and visualize. Because these concepts translate naturally to graphs, many applications use heuristics which interact directly with the graph. For instance, community detection algorithms use graph connectivity to detect groups within a social network.

Computer networks are another application where a graph representation is commonly used. Graphs can be used to model the physical or logical connectivity between computers on a network [1]. Alternatively, a graph might be used to represent the communication between networked machines [2]. Multiple applications in this work use graphs to capture how users authenticate to access a network's resources.

There are many examples of graph algorithms being applied to computer networks. Minimal spanning tree heuristics are used to control network routing to avoid problematic cycles [3]. Graph partitioning methods are used to segment large computer networks to make it difficult for adversaries to penetrate the network [4]. The dependencies for exploiting system vulnerabilities are modeled using attack graphs that highlight the possible paths attackers can take to compromise network resources [5].

The advantage of utilizing a graph representation is that it is a simple matter to apply existing graph algorithms to these applications without the need for problem-specific knowledge.

Some of this problem-specific information can, however, be incorporated to increase the accuracy of the graph model. For instance, network links that see higher traffic volume can be distinguished using weighted edges. The relative value of a network asset can be represented using vertex weights. Since vertex and edge weights are very common in graph models, many existing algorithms can already take them into account. Other problem-specific information can be included in a graph model using vertex or edge attributes, but traditional graph algorithms will typically ignore these features.

Customized heuristics can be developed that leverage additional problem-specific knowledge. Additionally, it is possible to achieve improved algorithm performance by exploiting graph characteristics that are common in an application area. Automated heuristic selection techniques can be used to select the most appropriate algorithm from a set of candidates [6]. Improved performance with this approach relies on having a good set of high-quality heuristics. Instead, domain expertise can be exploited to design novel customized heuristics.

The process of designing new heuristics can be accomplished manually, but this can be difficult and time consuming. Alternatively, generative hyper-heuristic techniques, such as genetic programming, can be used to automatically design novel algorithms [7]. The automated design capability of hyper-heuristics has been demonstrated in a variety of problem domains, but applications involving graph algorithms specifically are relatively new.

One possible explanation for the lack of adoption of hyper-heuristics in graph-based applications is the relative complexity of the desired solutions. Hyper-heuristics are often used to optimize relatively simple functions such as algebraic or otherwise symbolic expressions. These functions are often critical heuristic components of more involved systems. For example, a hyper-heuristic search was used to optimize the variable selection heuristic used for a type of SAT solving algorithm [8].

Hyper-heuristics can have difficulty generating larger programs that have the same level of complexity seen in human-designed solutions. Including the algorithmic building blocks needed to generate the fine details of a complex algorithm typically results in an unfeasibly large search space. This can be counteracted to some extend by including a smaller set of more

complex algorithmic components, but this can limit the flexibility of the heuristic-search to perform fine tuning.

The hyper-heuristic applications described in this dissertation attempt to tackle this problem in order to allow the search to find high-quality solutions to complex real-world problems. Initially, this is done through careful cultivation of a set of graph-based algorithmic components that incorporate critical functionality without resulting in an impossibly large search space. Additionally, a novel method of dynamically altering the granularity of the algorithmic components is investigated that has the potential to enable greater search flexibility while mitigating against the explosion in solution space.

This dissertation describes a variety of research projects involving the application of evolutionary and hyper-heuristic search techniques to optimize graph-based solutions to real-world cybersecurity problems. Chapter 2 presents a method of network segmentation that mitigates potential network intrusion from adversaries utilizing credential theft attacks; this work was published in IEEE Transactions on Dependable and Secure Computing [9]. The work presented in Chapter 3 improves upon previous work that uses genetic programming to automate the design of random graph models. Chapter 4 presents the potential performance gain of automating the design of novel graph partitioning heuristics. Chapters 3 and 4 were published in 2016 IEEE Symposium Series on Computational Intelligence [10, 11].

Chapter 5 details a generative hyper-heuristic search being used to automate the design of novel network security metrics; this work was published in the 2018 Proceedings of the Genetic and Evolutionary Computation Conference Companion [12]. The automated design of random graph algorithms was extended to dynamic graph applications in Chapter 6. The work presented in Chapter 6 was published in the 2019 Proceedings of the Genetic and Evolutionary Computation Conference Companion [13].

Chapter 7 is an extension of the work published in the 2019 Proceedings of the Genetic and Evolutionary Computation Conference Companion which covers the application of hyper-heuristic techniques to automate the design of tailored link prediction heuristics for anomaly detection applications in enterprise network security [14]. This extended chapter improves upon the previous work by incorporating a novel method for altering the granularity of the

hyper-heuristic search during evolution and investigates several mechanisms for automating this process. This chapter is currently being prepared for submission for publication in the IEEE Transactions on Dependable and Secure Computing. In Chapter 8, this dynamic granularity control process is also applied to the automated design of multi-level graph partitioning algorithms described in Chapter 4, this time with specific emphasis on computer network segmentation applications. This chapter is being prepared for submission for publication in the 2020 Proceedings of the Genetic and Evolutionary Computation Conference. Finally, Chapter 9 summarizes the work presented in this dissertation and offers some concluding remarks.

Chapter 2

Evolving Bipartite Authentication Graph Partitions

As large scale enterprise computer networks become more ubiquitous, finding the appropriate balance between user convenience and user access control is an increasingly challenging proposition. Suboptimal partitioning of users' access and available services contributes to the vulnerability of enterprise networks. Previous edge-cut partitioning methods unduly restrict users' access to network resources. This paper introduces a novel method of network partitioning superior to the current state-of-the-art which minimizes user impact by providing alternate avenues for access that reduce vulnerability. Networks are modeled as bipartite authentication access graphs and a multi-objective evolutionary algorithm is used to simultaneously minimize the size of large connected components while minimizing overall restrictions on network users. Results are presented on a real world data set that demonstrate the effectiveness of the introduced method compared to previous naive methods.

2.1 Introduction

Large scale enterprise computer networks are becoming ubiquitous and are increasing in complexity. The ease with which users access network resources directly impacts their productivity. Centralized *single-sign-on* (SSO) systems, such as Kerberos [15], allow organizations to manage access control on a large scale. However, due to the mechanics of these systems, if user access is granted without consideration for the security of the network as a whole, large portions of the network can become vulnerable to attack by an adversary that can compromise a user's credentials.

The credentials used to access a computer are often stored in a specialized cache on that machine. A variety of methods exist which allow an adversary to retrieve these credentials from a compromised computer [4]. Once the credentials have been obtained, they can be used to access and compromise other computers on the network. This entire process can be applied repeatedly, allowing an intruder to continue to traverse a growing portion of the network.

The most notorious example of exploiting stolen credentials, known as *pass-the-hash*, abuses the weakness of unsalted password hashes in older networks using Windows NT LAN Manager [16]. However, similar principles make this type of attack possible on modern systems as well, such as Kerberos [17]. Dunagan et al. refer to the process of repeatedly using stolen credentials to access additional computers as an *identity snowball* attack [4].

Segmenting the network by the accounts that are authorized to access them improves the resilience of the entire network against such attacks. Modeling a computer network as a graph makes it possible to employ graph partitioning algorithms to achieve this end, traditionally by selecting a set of edges to remove. While effective, those methods do not take advantage of the nature of the computer networks these graphs represent.

This work improves upon network partitioning by using a representation that considers the nature of user authentication. This allows for an approach that more specifically solves for the problem of user access control instead of the more abstract problem of graph partitioning. Computer networks are modeled as graphs consisting of user and computer nodes connected by edges corresponding to authentication events. These networks are partitioned using a technique that leverages a traditional graph partitioning heuristic along with an evolutionary algorithm to produce solutions that protect the network from credential theft attacks without unnecessarily restricting user access to network resources.

2.2 Background

2.2.1 Graph Partitioning

Given a graph $G = (V, E)$ with the set of vertices V and the set of edges E, a graph partitioning divides the vertices in G into smaller subsets of V. In many applications, desirable partitions

are those for which the number (or sum of the weights) of edges in E that connect vertices in different subsets is small. This is usually because the partition is applied to the graph by removing any such edges to disconnect the graph into separate components. Each edge removal typically incurs some cost depending on the specific application, and this overall cost should be minimized. Previous research has shown that partitioning a graph by removing the minimum number (or weight) of edges, even when allowing unbalanced partitions, is an NP-hard problem [18]. For many applications, it is sufficient to use heuristics to approximate an optimal graph partition. For these instances, software has been developed to quickly find low cost partitions. The METIS graph partitioning package is an example of such an implementation [19].

2.2.2 Evolutionary Algorithms

An *Evolutionary Algorithm* (EA) is a biologically inspired generate-and-test, black-box search technique [20]. A population of solutions to a specific problem is randomly generated. During *recombination*, this population is used to generate offspring by combining portions of donor parent solutions. Offspring are typically subjected to some form of *mutation*, which is capable of introducing entirely new genetic information, enabling exploration of the search space. Some portion of the population is selected to survive, or continue on to the next generation where the entire process is repeated until some termination criteria is met. The *fitness* of a solution represents how well it solves the problem at hand. Selection pressure, which generally involves favoring higher fitness when selecting solutions to produce offspring or survive into later generations, encourages exploitation of genetic information known to contribute to higher fitness.

In a *Multi-Objective Evolutionary Algorithm* (MOEA), a single fitness value is replaced by a set of (often competing) objective values. MOEAs, such as the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [21], compare relative solution quality using a dominance relation instead of a simple fitness comparison. One solution dominates another if it is at least as good for all objectives and strictly superior for at least one objective. Instead of returning the

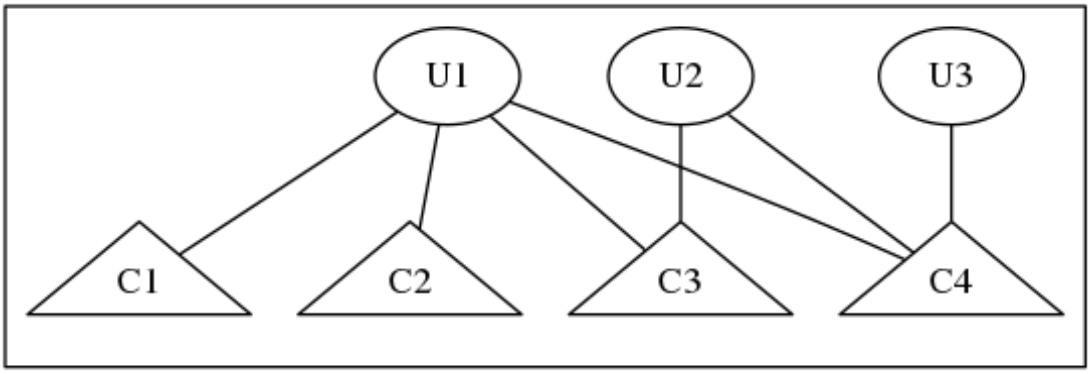


Figure 2.1: Example BAG with users U1, U2 and U3 and computers C1, C2, C3 and C4. An edge represents an authentication event between a user and a computer.

single best solution found, MOEAs typically return the set of the best non-dominated solutions; this set is referred to as the *Pareto frontier*.

2.3 Bipartite Authentication Graphs

The computers on a network and the user accounts that access them can be naturally represented as two independent sets of nodes in a bipartite authentication graph (BAG) [22]. An edge in this graph connects a user node to a computer node and represents an occurrence where the user's authentication credentials are used to access the computer. This access can be direct (e.g., a user logging into a workstation) or indirect (e.g., through SSH or a remote desktop session).

Example 1 *The BAG in Figure 2.1 contains the set of user nodes* $\{U1, U2, U3\}$*, the set of computer nodes* $\{C1, C2, C3, C4\}$ *and the set of authentication edges* $\{(U1, C1), (U1, C2), (U1, C3), (U1, C4), (U2, C3), (U2, C4), (U3, C4)\}$.

This graph representation makes it possible to identify the portions of a network which are vulnerable to credential theft attacks. For example, if the computer $C1$ in Figure 2.1 is compromised, the credentials for user $U1$ could be stolen. The existing edges of the BAG indicate that the credentials for user $U1$ can also be used to access computers $C2$, $C3$, and $C4$. As a result, an adversary armed with the stolen credentials for user $U1$ would also be able to gain access to these additional computers. The same exploits used to steal credentials from

computer $C1$ can then be repeated on these new targets allowing the intruder to continue to traverse the network.

Under normal circumstances, a computer's cache would only contain a subset of the credentials used to access the machine due to limits on the cache size or credentials being periodically removed. Since it is not always possible to determine which credentials are present at a particular time, assuming all the previously used credentials are still in the cache gives an upper limit on the potential risk when the machine is compromised. If the edges of a BAG incident to a given computer represent the authentication credentials assumed to be currently stored in that computer's cache, then upon compromise, the adversary could have access to the credentials of all adjacent user nodes in the BAG. If any of these users are also adjacent to other computer nodes in the BAG, then their stolen credentials can be used by the adversary to access those machines. By repeating this process, the adversary can continue traversing the connected component of the BAG compromising a growing portion of the network.

2.3.1 BAG Partitioning

Large connected components in the BAG represent a greater potential for the damage that can be done with repetitive credential theft attacks. Reducing the size of the connected components would limit the assets an adversary could gain access to after an initial compromise. One straightforward method of disconnecting components in a graph is to use a traditional edge removal partition algorithm. Removing a set of edges to split the large connected components of a BAG translates to revoking the ability of certain users to access a subset of their adjacent computers.

Example 2 *Figure 2.2 shows the BAG in Figure 2.1 after applying the partition:* $edgesToRemove = \{(U1, C3), (U1, C4)\}$*, which separates the BAG into the following partitions:* $\{U1, C1, C2\}, \{U2, U3, C3, C4\}$*. This partition is translated to the network by revoking user* $U1$*'s access to computers* $C3$ *and* $C4$*. If an adversary compromises the computer* $C1$ *and steals the credentials for user* $U1$*, the intruder will no longer be able to use those stolen credentials to access* $C3$ *and* $C4$ *as a result of this partitioning.*

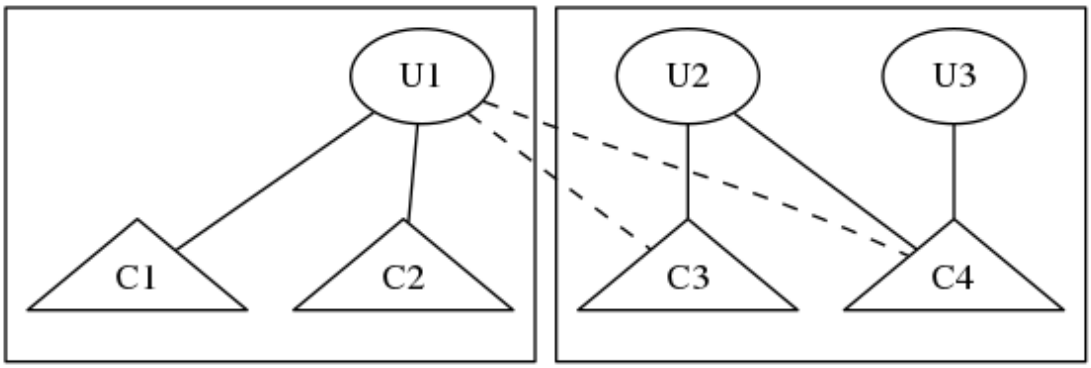


Figure 2.2: Partitioning of the BAG in Figure 2.1 into two connected components by removing two edges. Dashed lines are edges that have been removed during partitioning. Rectangle borders surround connected components.

Assuming the user had a good reason for accessing the computer, this restriction could prevent the user from utilizing necessary resources. It is possible that the user could be given access to a suitable replacement computer, but this would require information about available network resources as well as the purpose of the original access. An alternative solution would be to give that user a second account with separate credentials that would be used to access a subset of the computers they use. In the BAG, this would split a user node into two nodes each connected to a subset of the computers to which the original node was adjacent.

Example 3 *Figure 2.3 shows the BAG in Figure 2.1 after splitting the user node* $U1$ *with the following scheme:* $userSplits = \{U1 : \{C1, C2\}, \{C3, C4\}\}$*, which separates the BAG into the following partitions:* $\{U1a, C1, C2\}, \{U1b, U2, U3, C3, C4\}$*. This partition is translated to the network by replacing user* $U1$*'s authentication credentials with two new sets of credentials. With the first set of credentials (*$U1a$*),* $U1$ *can access computers* $C1$ *and* $C2$*.* $U1$ *can use the second set (*$U1b$*) to access computers* $C3$ *and* $C4$*.*

This technique can be used to disconnect a critical path between two large components in the network while still allowing the user to access the needed computer resources. The process can also be extended to split the user into more than two accounts, although this could quickly become cumbersome for a user who accesses more than the average number of computers, such as a network administrator. In order to prevent partition solutions that require a user to manage

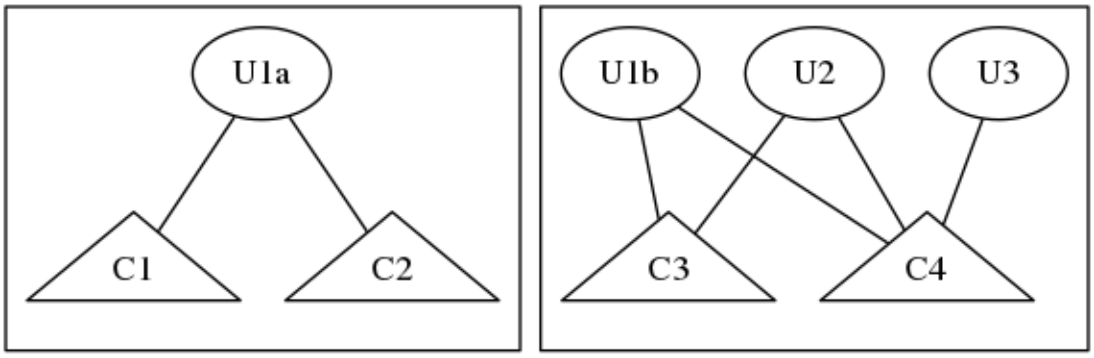


Figure 2.3: Partitioning of the BAG in Figure 2.1 by splitting user U1 into two user nodes U1a and U1b.

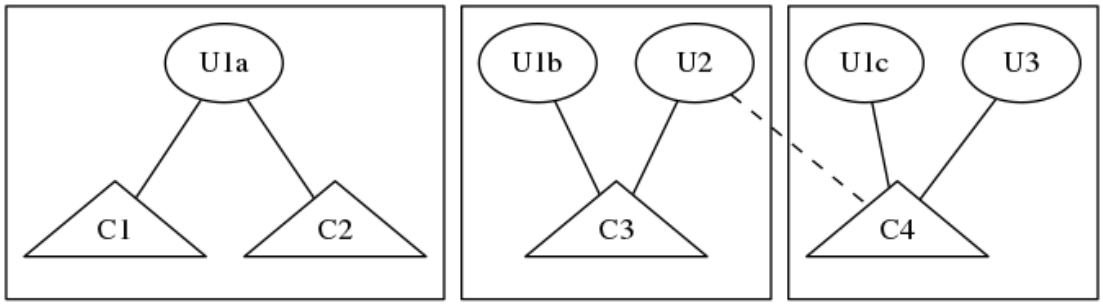


Figure 2.4: Partitioning of the BAG in Figure 2.1 by splitting user U1 twice and removing an edge.

an unreasonable number of credentials, a limit on the number of splits per user node can be specified. This limit would vary depending on the application, or could be omitted entirely to allow unlimited user node splitting. If this limit is required by an application, some number of edge removals may still be warranted to ensure disconnecting a critical path in the BAG.

Example 4 *Figure 2.4 shows the BAG in Figure 2.1 after applying the following partition:* $edgesToRemove = \{(U2, C4)\}$, $userSplits = \{U1 : \{\{C1, C2\}, \{C3\}, \{C4\}\}\}$*, separating the BAG into the following partitions:* $\{U1a, C1, C2\}$, $\{U1b, U2, C3\}$, $\{U1c, U3, C4\}$.

For the purpose of partitioning the BAG, a similar node splitting process can also be performed on the computer nodes. Translating such node splits to the real network is not as simple as giving a user a second set of authentication credentials, however. A computer node split could be implemented by adding another computer to the network and requiring a subset of the original computer's users to instead make use of the new machine. If the computer node

in question represents a server running virtual machines, a split could also take the form of an additional instance of the virtual environment being run with access restrictions to each instance. This approach might be more feasible because it does not require the allocation of additional hardware, but the increase in simultaneous virtual environments would introduce further computational overhead. Because these implementation methods might not be practical for a particular computer network, this research focuses on partitions that are limited to splitting user nodes.

Removing an edge and splitting a user node in the BAG both translate to an impact on a user's ability to efficiently access the resources they need to perform their work. These measures should only be applied if they make a significant impact on the size of the connected components in the graph. An ideal solution will strike a balance between the impact on the users and the reduction in the size of the connected components.

Since the security needs of a particular network vary based on its purpose, presenting a single partitioning solution might not be adequate. An approach that produces multiple possible solutions with various objective trade-off values would allow a network administrator to choose a partition solution that meets their needs in terms of security and efficiency. Multi-objective evolutionary algorithms, such as NSGA-II [21], excel at this type of problem where it is not possible to determine the relative value between two or more conflicting objectives a priori.

2.3.2 BAG Partition Application

Conceptually, applying a BAG partition solution to the actual network is a straightforward process. When a user node split is prescribed by a partition, the associated account is replaced by multiple new accounts, each with access to a subset of that user's originally accessed machines. Should an adversary compromise one of these computers and obtain the user's credentials, they will not be able to use those credentials to gain access to computers in a different subset, possibly disconnecting their path to additional network assets.

An obvious way to translate edges removed by the BAG partition is to revoke the associated user's access to a computer entirely. However, this drastic solution may not always be

practical. Compared to traditional edge removal partitioning algorithms, partition solutions involving user node splits tend to require dramatically fewer edge removals. As a result, it is more feasible to explore alternatives when edge removals prove necessary. The suggested edge removals indicate critical paths between connected components in the BAG. If these connections cannot be removed for practical reasons, they could instead be targeted with high-fidelity traffic monitors to detect adversaries traversing the network. Another option, as previously discussed, is to allocate a suitable replacement computer for the user that results in smaller connected components. Alternatively, a small number of user nodes, such as those corresponding to network administrators, could be split more times than is normally allowed for a regular user.

2.4 Related Work

This work is related to other methods which use a graph representation of possible avenues of attack against network vulnerabilities. In particular, this work directly extends previous research that introduced BAGs and discussed how they can be used to model the vulnerability of a network when adversaries are able to steal user credentials [23]. Previous related work in combating this type of attack usually takes one of two forms: identifying and responding to network intrusion [24], or partitioning the network by controlling user access to limit the access an adversary can gain with a set of stolen credentials [4]. This work focuses on the latter by creating partitions of the graph network representation and translating these partitions into network policies.

Attack graphs have traditionally been used to visualize potential paths of attack that exploit a variety of system vulnerabilities in a network [5]. Because attack graphs highlight network vulnerabilities, they serve as an important tool for identifying when an attack is taking place as well as preventing possible attacks altogether. Their value as a security tool has prompted work in automatically generating them from network data [25, 26], as well as visualizing them for large networks [27]. A BAG can be viewed as an attack graph that focuses on credential stealing attack potential. Because of their singular purpose, BAGs can be constructed without requiring information about the individual vulnerabilities present on the networked systems.

Since this work is concentrated on limiting the damage of these types of attacks, BAGs are used in favor of traditional attack graphs.

The security metrics by which attack graphs are measured have also received a lot of attention. Many look at characteristics of possible paths in the attack graph that reach some predesignated goal. Examples include the length of the shortest path [5], the number of paths [28], and the average length of paths [29]. These methods are difficult to apply to the problems considered in this research, because the goal of the attacks being considered is not reaching an individual machine, but compromising the largest possible portion of the network. A more relevant measure is the Network Compromise Percentage (NCP) metric described in [30], which calculates the percentage of network assets that can be compromised by an attack.

Previous work introduced the use of BAGs and examined the effect of removing high degree nodes on the size of connected components [22]. This simple approach can reduce the size of the connected components, but is difficult to translate to an actual computer network, since the removal of the node would correspond to disabling a user account or removing a computer from the network. More sophisticated approaches exist, such as Heat-ray, which finds sparse cut partitions in an attack graph [4]. This method leverages traditional attack graphs as well as feedback from a network administrator to suggest security configuration changes that are easier to apply to the actual network. The solution presented in this work does not rely on repeated interaction with an administrator, but instead produces a variety of possible security configuration solutions. The administrator can then select from these options the solution that best suits the specific security needs of their network. Also, since this approach does not rely on traditional attack graphs, it can be applied even when information about present system vulnerabilities is not available and cannot be collected.

Instead of limiting an adversary's ability to traverse the network through the use of partitioning, intrusion detection systems aim to identify the adversary's presence on a network as the attack is taking place [24]. This approach generally relies on either matching known attack signatures [31] or by identifying abnormal behavior by comparison to normal or known legitimate activity [32]. There are examples of evolutionary algorithms being applied to improve the

performance of intrusion detection systems [33]. The work presented in this paper also leverages the strengths of an evolutionary algorithm, but serves as more of a mitigation technique that does not rely on being able to identify an attack in real time to be effective.

This work relies on advances in efficient graph partitioning. Due to the large size of the graphs involved, finding the optimal partition is not feasible. Multi-level graph partitioning is a widely used approach to approximating low-cost graph partitions [34]. The process approximates the input graph using a smaller, easier to partition version, then maps the simple partition solution back to the original graph. Several well-known graph partition software packages implement multi-level schemes, such as METIS [19], JOSTLE [35], Scotch [36], and DiBaP [37]. Unlike these general purpose graph partition solutions, this work leverages problem specific knowledge to produce superior partitions specifically for authentication graphs.

There are many examples of evolutionary computation techniques being used to find approximate minimum graph partitions [38, 39]. The Karlsruhe Fast Flow Partitioner Evolutionary (KaFFPaE) leverages the inherent parallelizability of evolutionary algorithms to evolve graph partitions on a distributed system [40]. Soper et al. introduced an evolutionary search algorithm that makes use of a multilevel heuristic for crossover to generate high quality graph partitions [41]. Benlic and Hao developed a multilevel memetic algorithm for the k-way graph partitioning problem [42]. These approaches are similar in that they each assess the quality of a partition solely by determining the cost of removing the necessary edges. The work presented in this paper considers this cost as well, along with the cost of splitting user nodes. Unlike the mentioned approaches, this work does not require the number of desired partitions to be specified a priori. Instead, a multi-objective evolutionary algorithm is used to produce a set of partition solutions with a variety of partition sizes while minimizing the cost of applying the partition.

The approach presented in this work is not intended to be a stand-alone solution to computer network security. Instead, it is intended to provide an additional component in a multi-layer defense system. It should be combined with traditional network hardening practices, such as firewall configuration, software security patch maintenance, as well as utilizing anti-virus

and intrusion detection systems. Even with these implementations in place, not all compromises can be prevented. It is in these cases where partitioning the network by user access can mitigate the damage potential of credential theft attacks when they do occur. Modeling the network with a BAG allows this partitioning to be done and only requires information on authentication events, which can be collected easily on enterprise networks employing a single-sign-on system. Using a multi-objective approach means that the network administrator will be presented with an assortment of security configuration change solutions, from which an option can be implemented that meets the security needs of their particular network.

2.5 Methodology

Three methods of partitioning BAGs are considered. The first is a naive method previously introduced that iteratively removes the highest degree node from the graph [22]. The second method uses the METIS software package [19] to find a variety of edge-cut partitions of the input BAG. The final approach leverages the strength of the METIS partition algorithm, but improves upon the result by using an evolutionary algorithm to evolve partition solutions consisting of edge removals and user splits from a population of randomly generated partitions.

2.5.1 Naive Iterative Node Removal

The naive algorithm defined in Algorithm 2 starts with the input BAG and iteratively selects the highest degree node and removes it from the BAG, which also removes all edges incident to that node.

Algorithm 1 Get Computer Component Cap: Calculates the number of computer nodes in each connected component and returns the maximum. Note: $computerCount(component)$ returns the number of computer nodes in $component$.

```
procedure GCCC(BAG)
    CCC ← 0
    for component ∈ connectedComponents(BAG) do
        if computerCount(component) > CCC then
            CCC ← computers(component)
    return CCC
```

Algorithm 2 Naive Partition: Repeatedly selects and removes a node of maximum degree from the graph until the maximum number of computer nodes in any connected component is less than $computerComponentLimit$.

procedure NAIVEPARTITION(BAG, $computerComponentLimit$)
  while GCCC(BAG) $>$ $computerComponentLimit$ **do**
    $n \leftarrow$ highest degree node in BAG
    $BAG.removeNode(n)$

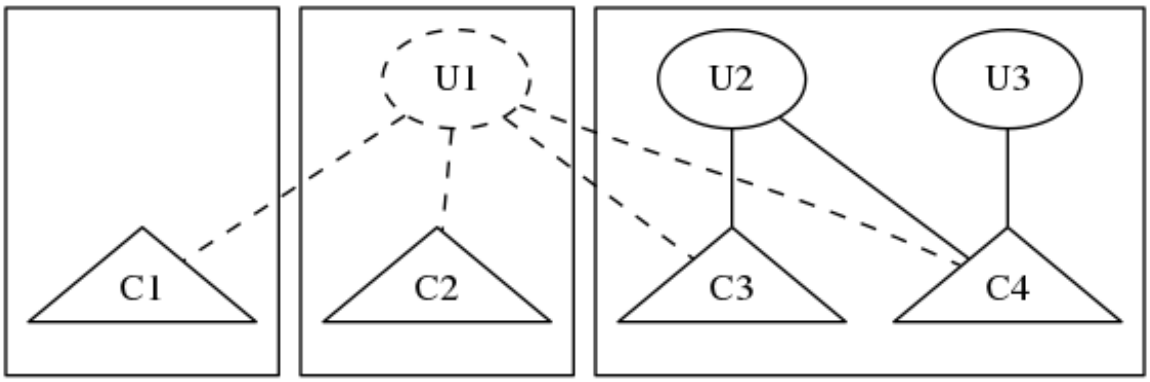


Figure 2.5: The BAG that results from applying one iteration of Algorithm 2 to the BAG in Figure 2.1. The dashed lines indicate components which have been removed by the algorithm. The user node $U1$ has been removed along with its incident edges.

Example 5 *Applying Algorithm 2 to the BAG from Figure 2.1 with a* $computerComponentLimit$ *of* 2 *will select the highest degree node (user node* $U1$*) for removal along with all of its incident edges. The maximum number of computer nodes in any connected component will then be* 2 *and the algorithm will terminate. Figure 2.5 shows the result of this process.*

2.5.2 METIS

Algorithm 3 uses METIS' k-way partitioning to partition the input BAG. The algorithm assigns each node in the BAG a partition label ranging from 1 to k. Any edge that connects nodes that differ in partition labels is removed unless doing so would completely disconnect a user from the graph. A set of partition solutions are created by using a variety of k values to partition the same BAG.

Example 6 *Figure 2.6 shows a possible partitioning of the BAG from Figure 2.1 using the method described in Algorithm 3 for a* k *value of* 3*. The labels assigned by METIS are shown*

Algorithm 3 METIS Partition: Partitions an input graph using METIS k-way partitioning.

procedure METISPARTITION(BAG, k)
    $labels \leftarrow METIS.kWayPartition(BAG, k)$
    for $(user, computer) \in BAG.edges$ **do**
        if $labels[user] \neq labels[computer] \wedge degree(user) > 1$ **then**
            $BAG.removeEdge(user, computer)$

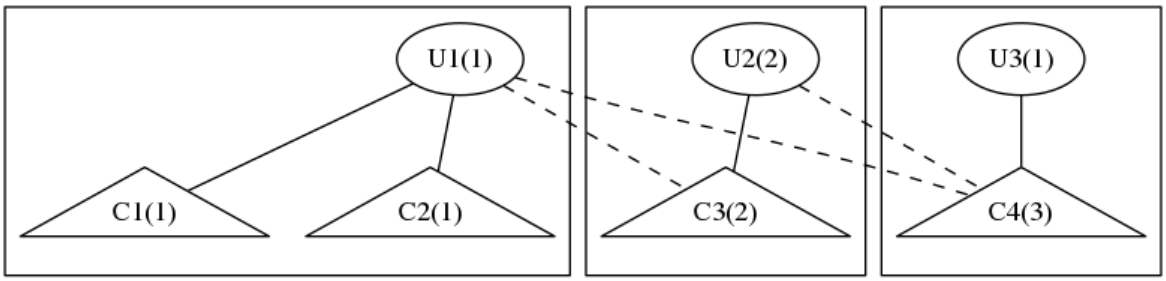


Figure 2.6: A possible result of applying Algorithm 3 to the BAG in Figure 2.1. The labels assigned by METIS are shown following the original node labels. A dashed line indicates an edge that has been removed because the endpoints have differing labels. The edge $(U3, C4)$ is not removed because this would completely disconnect user node $U3$.

following the node labels. The edges $(U1, C3)$*,* $(U1, C4)$ *and* $(U2, C4)$ *are removed by the algorithm, because nodes* $U1$*,* $U2$ *and* $C4$ *all have different label assignments (*1*,* 2 *and* 3*, respectively). For the purposes of this example, user node* $U3$ *was assigned a label of* 1 *to illustrate that it will not be disconnected from computer node* $C4$*, because this would completely disconnect* $U3$ *from the BAG.*

2.5.3 Evolutionary Algorithm

A population of partition solutions is evolved using the Nondominated Sorting Genetic Algorithm II (NSGA-II). NSGA-II has been applied to multi-objective graph partitioning previously, with promising results [39].

Initialization: For each individual partition solution, a set of user nodes to be split is sampled from all user nodes uniformly. A k value is randomly generated from a configurable range of possible partition labels.

Recombination: Two new child partition solutions are created from two parent solutions. One child receives its k value from the first parent and the other child copies the value from the

second parent. The $usersToSplit$ sets for the children are created by iterating over an ordered list of user nodes. This list is divided by a configurable number of crossover points that are determined randomly. The pair of children is matched to the pair of parents so that the first parent contributes to the $usersToSplit$ set of one child, and the second parent contributes to the other. For each user node iteration, if the donor parent's $usersToSplit$ set contains that user node, the user node is also added to the associated child's $usersToSplit$ set. Whenever a crossover point is encountered, the matching of parent to child solutions is reversed so that each child begins copying the $usersToSplit$ set of the alternate parent.

Example 7 *Crossover recombination of* $usersToSplit$ *attributes with six users and two randomly selected crossover points. The vertical bars indicate crossover points.*

$BAG.users$	$U1, U2$	$U3, U4$	$U5, U6$
$parent1.usersToSplit$	$U2$	$U3$	$U6$
$parent2.usersToSplit$	$U1$	$U4$	$U5$
$child1.usersToSplit$	$U2$	$U4$	$U6$
$child2.usersToSplit$	$U1$	$U3$	$U5$

Mutation: A new partition solution has a configurable probabilistic chance to randomly increment or decrement its k value within the valid limit. Each user node also has a chance of being mutated by adding it to the solution's $usersToSplit$ set if it is not already present or removing it otherwise.

Evaluation: In order to determine the quality of a solution, the partition is used as input to the procedure defined in Algorithm 4. A mapping is created that stores the adjacent computer nodes for each user in the solution's set of user nodes ($usersToSplit$). These user nodes are then removed from the graph and METIS' k-way partitioning algorithm is applied to resulting graph as described in Section 2.5.2. For each removed user node, the list of adjacent computers is grouped by the partition assignments produced by the METIS partition. For each resulting group, a new user node is added and then connected to the computer nodes in that group. If the number of groups exceeds the maximum splits per user, the smallest groups are simply truncated resulting in additional edge removals.

Termination: The consolidation ratio metric described in [43] is used to detect convergence. An archive of the non-dominated solutions discovered by NSGA-II is updated periodically during evolution. The update is performed every ten generations to reduce the likelihood of premature convergence detection, as suggested by the authors of the method. When the percentage of solutions that remain non-dominated in the archive exceeds a configurable limit, the evolution is terminated.

Algorithm 4 Partitions BAG using a set of users. Stores a mapping of the computers adjacent to users in $usersToSplit$ then removes these user nodes. Uses METIS to partition the resulting graph. The adjacent computer lists are then partitioned by $splitUserNode$. The user nodes in $usersToSplit$ are then added back to the graph as split user nodes, connecting them to the original user's adjacent computer nodes in such a way that minimizes the size of connected components in the BAG.

procedure EVOPARTITION(BAG, k, $usersToSplit$, $maxUserSplits$)
    $adjacentComputers \leftarrow Dictionary()$
    for $user \in usersToSplit$ **do**
        $adjacentComputers[user] \leftarrow BAG.neighbors(user)$
        $BAG.removeNode(user)$
    $labels \leftarrow METIS.kWayPartition(BAG, k)$
    for $(user, computer) \in BAG.edges$ **do**
        if $labels[user] \neq labels[computer] \wedge degree(user) > 1$ **then**
            $G.removeEdge(user, computer)$
    $computerPartitions \leftarrow splitUserNodes(BAG, usersToSplit, adjacentComputers, maxUserSplits)$
    for $user \in usersToSplit$ **do**
        $partitionList \leftarrow computerPartitions[user]$
        for $i \in 1..|partitionList|$ **do**
            $partition_i \leftarrow partitionList[i]$
            $user_i \leftarrow newNode()$
            $BAG.addNode(user_i)$
            for $computer \in partition_i$ **do**
                $BAG.addEdge(user_i, computer)$

Objectives: Two objectives are used to compare the quality of evolved partition solutions:

User Impact: The sum of the weight of the edges removed from the BAG, plus the total number of splits performed on user nodes in the BAG. Splitting a user into two nodes counts as a single user split, splitting a user into three nodes counts as two user splits, etc. In a practical application, it might not be appropriate to simply sum these two values. Instead, an administrator might choose to make them separate objectives, or scale the two values to indicate

Algorithm 5 Split User Node: Partitions adjacent computer nodes using the labels assigned by METIS.

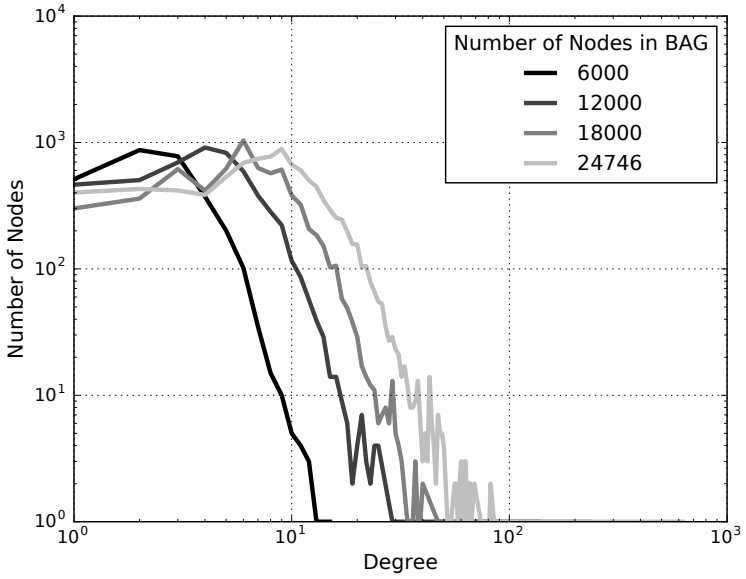
procedure SPLITUSERNODE(BAG, $usersToSplit$, $adjacentComputers$, $maxUserSplits$)
    $partitions \leftarrow Dictionary()$
    for $user \in usersToSplit$ **do**
        $partitions[user] \leftarrow adjacentComputers[user]$ partitioned by METIS labels in BAG
        Sort $partitions[user]$ in decreasing order by the size of the partitions
        Truncate $partitions[user]$ to length $maxUserSplits$
    return $partitions$

the relative difficulty of applying one option over the other. The simplest approach of using the unweighted sum of the two values is chosen in this work to make the results easier to interpret.

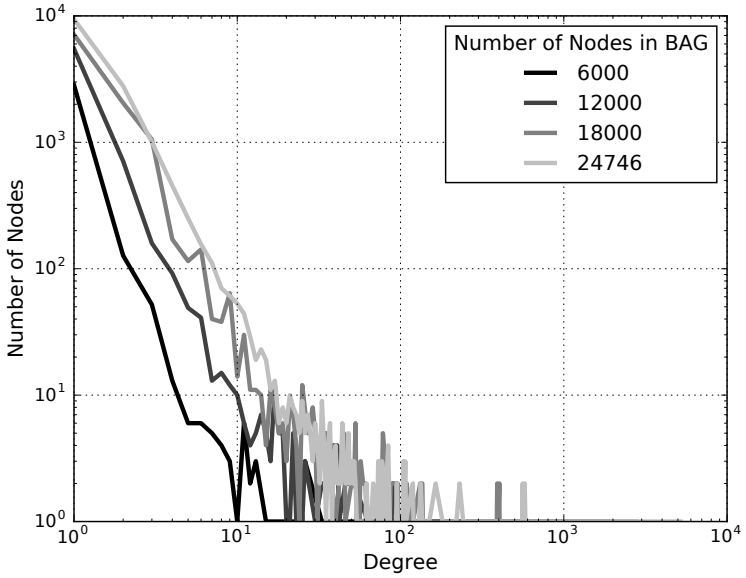
Average Network Compromise Percentage: A computer node's Network Compromise Percentage (NCP) is the number of computer nodes in the same connected component, divided by the number of computer nodes in the entire BAG; this corresponds to the percentage of computers that can be reached if the initial computer is compromised and used as a launching point for an unmitigated credential theft attack. Note that since each computer in a connected component has the same NCP value, they can be calculated simultaneously for entire components. The Average Network Compromise Percentage objective value for a solution is the average NCP of all of the computer nodes in the BAG after the partition is applied.

2.6 Experiment

Authentication data from the network at Los Alamos National Laboratory (LANL) for a month of regular activity is used to construct a BAG [44]. The graph is shown in Figure 2.8a and contains 9,924 user nodes, 14,822 computer nodes and 106,693 authentication edges. The largest of the 201 connected components contains 9,724 of the user nodes, 14,608 of the computer nodes and 106,479 of the edges. To examine the impact of the input graph size on the quality of the partitioning, as well as demonstrate the generality of the approach, a series of BAGs are also randomly generated and partitioned. In order to ensure that these random graphs still resemble enterprise computer networks, they are generated using user and computer node degree distributions that are similar to those found in the LANL network data set. Figure 2.7 shows



(a) User Node Degrees



(b) Computer Node Degrees

Figure 2.7: Degree distribution comparison of full LANL network BAG and three smaller randomly generated BAGs.

just how similar these degree distributions are. The horizontal shift of the curves is the result of the increasing number of nodes in the BAG, which increases the number of nodes with specific degree values as well as the maximum possible degree.

The LANL network data set does not contain any information about the relative value of each computer asset or the importance of any given authentication event. For this reason,

all computer nodes and authentication edges are assumed to be equivalent and are given unit weight. If a network administrator did have some knowledge about the relative difficulty, or user impact of implementing an authentication removal, the weight of the corresponding authentication edge could be increased or decreased to discourage or encourage the likelihood of removing that edge, respectively. The additional effort of constructing a weighted BAG a priori can be minimized by assuming unit weight for most edges and only adjusting the weight of a small subset of important edges.

To investigate the impact of varying weight edges, a copy of the LANL network BAG is generated with random edge weights. Since METIS requires integer edge weights, the weights are taken uniform randomly from $\{0, 1, \ldots, 100\}$. When evaluating a partition solution, these edges weights are divided by 50 so that the mean edge weight is still one, making a direct comparison between the weighted and unweighted cases easier.

The parameters used for the NSGA-II approach are given in Table 2.1. To conform to standard NSGA-II, the value for Number of Offspring is set equal to the Population size and the Crossover Probability is set to 100%. The Minimum k value is set to 1 to allow solutions that only employ user node splits and do not use METIS to remove edges. The values for Population Size, Number of Crossover Points, Convergence Consolidation Ratio, Mutation Rate and Maximum k were selected using a random-restart steepest-ascent hill climbing search algorithm. The algorithm initializes with a randomly chosen value for each parameter and runs thirty experimental runs to convergence using those parameter values. To determine the relative quality of each configuration, the results they produce are compared using the technique described in [45], which is explained in Section 2.7.

For comparison, both the naive and METIS approaches are also used to partition each BAG. Unlike NSGA-II, these methods produce only a single partitioning solution each time they are run. To generate a population of solutions using the naive approach, the algorithm is repeatedly run with different values for the $computerComponentLimit$ input parameter. Initially, the $computerComponentLimit$ is set to the original BAG's Largest Computer Component Size value, then it is decremented for each repetition until it reaches 1. Alternatively, the METIS approach is repeatedly applied with increasing values of the k input parameter. The

Table 2.1: NSGA-II parameter values.

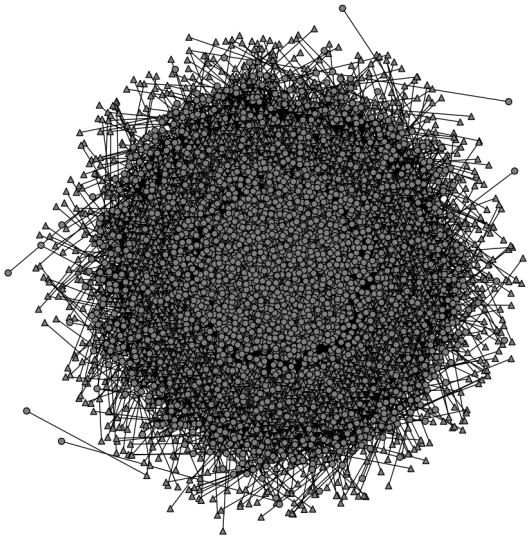
Parameter	**Value**
Population Size	61
Number of Offspring	61
Crossover Probability	1.0
Number of Crossover Points	2
Mutation Rate	10^{-9}
Convergence Consolidation Ratio	0.9
Minimum k	1
Maximum k	21
Maximum Splits per User	5

initial value of k is set to 2 to produce a bisection, then incremented with each iteration until further increases in k no longer change the output BAG (around $k = 2{,}500$).

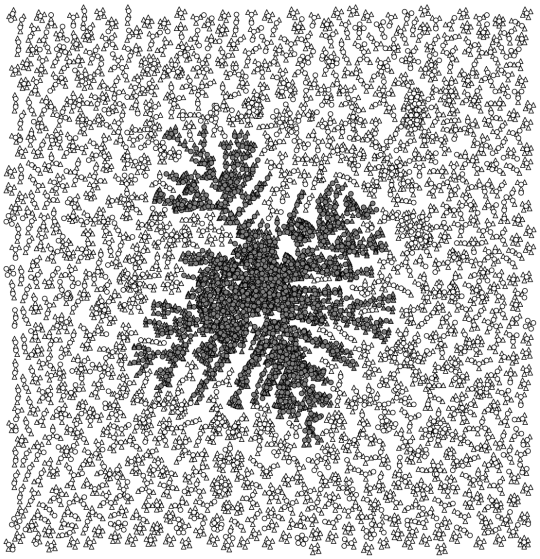
2.7 Results

Figure 2.8 shows the BAG created using the LANL network authentication data, along with three partitioned versions of the graph, one for each partitioning approach described in Section 2.5. Figure 2.9 shows the final objective values from 30 experimental runs using NSGA-II on a randomly generated BAG as well as the LANL network BAG. The horizontal and vertical axes measure the level of User Impact and the Average Network Compromise Percentage, respectively. The dashed line is a locally weighted regression line. The final objective values of a sample set of evolved partition solutions to the unweighted LANL network BAG from the NSGA-II process are listed in Table 2.2.

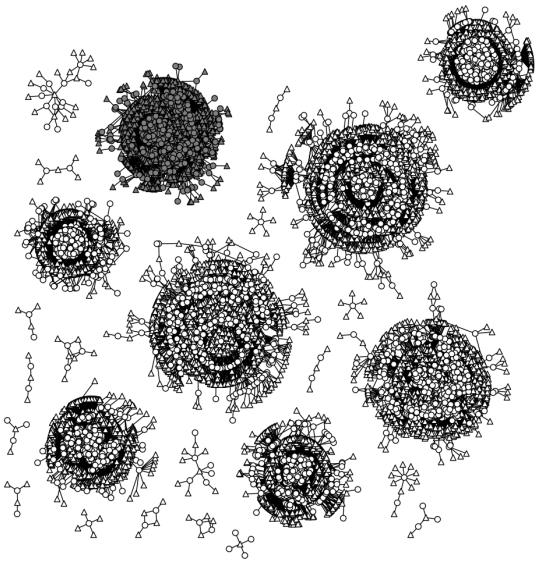
Figure 2.10 shows the improvements in the Pareto frontier during an example run of NSGA-II. Since both objectives are positive values being minimized, improvements travel towards the origin. A new Pareto frontier is added every ten generations and is a darker shade than the previous generations. The difference in areas between neighboring Pareto frontiers shows that a lot of improvement is found in early generations, but this improvement slows in later generations as the algorithm nears convergence.



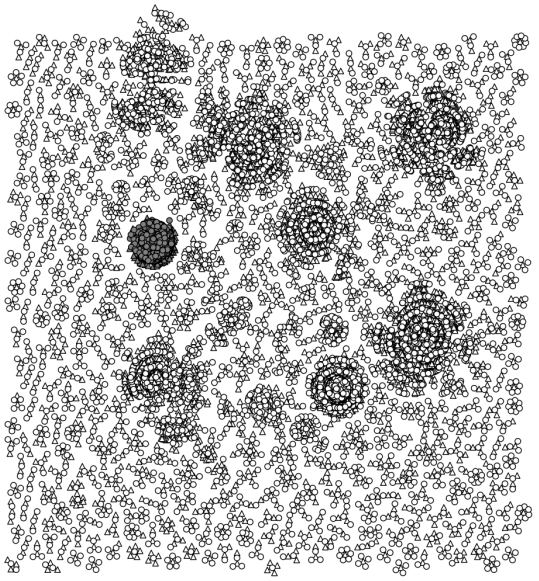
(a) BAG created from one month of LANL network authentication event data. The graph contains 9924 user nodes, 14822 computer nodes and 106693 authentication edges.



(b) BAG in Figure 2.8a partitioned using the naive iterative node removal method detailed in Section 2.5.1. The largest connected component contains 1998 computer nodes. 91226 authentication edges have been removed from the original BAG.

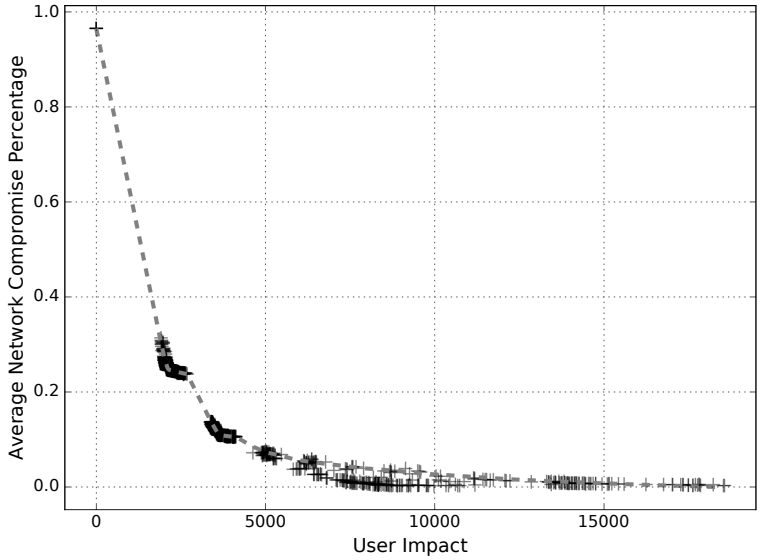


(c) BAG in Figure 2.8a partitioned using the METIS method detailed in Section 2.5.2 with a k value of 9. The largest connected component contains 1888 computer nodes. 43163 authentication edges have been removed from the original BAG.

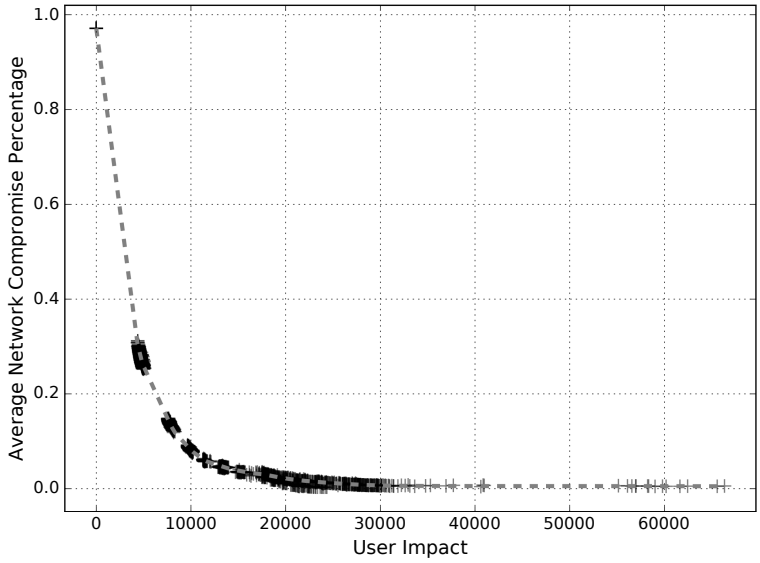


(d) BAG in Figure 2.8a partitioned using an example evolved partition as described in Section 2.5.3. The largest connected component contains 1830 computer nodes. The partitioning removes 2865 authentication edges and performs 15594 user node splits.

Figure 2.8: BAGs created from LANL network data as described in Section 2.6 and a sample partitioned BAG from each approach described in Section 2.5. User nodes are represented as circles and computer nodes are triangles. The shaded components indicate the connected component with the maximum number of computer nodes. Components of size smaller than 5 are omitted for clarity.



(a) Random BAG with 12000 nodes.



(b) Full LANL network BAG.

Figure 2.9: Objective values of evolved partition solutions from 30 experimental runs of NSGA-II for a randomly generated BAG as well as the full LANL network BAG. Each cross represents the objective values of a single solution. The dashed line is a locally weighted regression line.

Table 2.2: Characteristics of an example set of BAG partition solutions evolved by NSGA-II as described in Section 2.5.3. The shaded columns are the objective values used during the evolution.

User Impact	Edge Removal Cost	User Splits	Average Network Compromise Percentage
0	0	0	0.97133
4833	0	4833	0.30567
4902	0	4902	0.29611
4917	0	4917	0.29568
4943	0	4943	0.29489
5008	0	5008	0.28825
5083	0	5083	0.2859
5101	0	5101	0.28459
5133	0	5133	0.28128
5139	0	5139	0.27981
7971	0	7971	0.14948
8004	29	7975	0.14899
8084	6	8078	0.14725
8102	6	8096	0.14485
8163	0	8163	0.1436
8259	0	8259	0.14076
8333	0	8333	0.13981
8344	0	8344	0.137
8412	0	8412	0.13661
8477	0	8477	0.13448
10296	0	10296	0.090649
10344	0	10344	0.090331
10362	0	10362	0.089305
10531	94	10437	0.087049
10559	94	10465	0.085751
14962	1501	13461	0.036929
15091	1474	13617	0.035139
15662	1785	13877	0.035028
15692	1836	13856	0.033937
17169	1632	15537	0.033335
18573	2315	16258	0.024451
20717	3723	16994	0.012515
21882	4351	17531	0.0085489
21999	4608	17391	0.0080443
23610	5739	17871	0.0040842

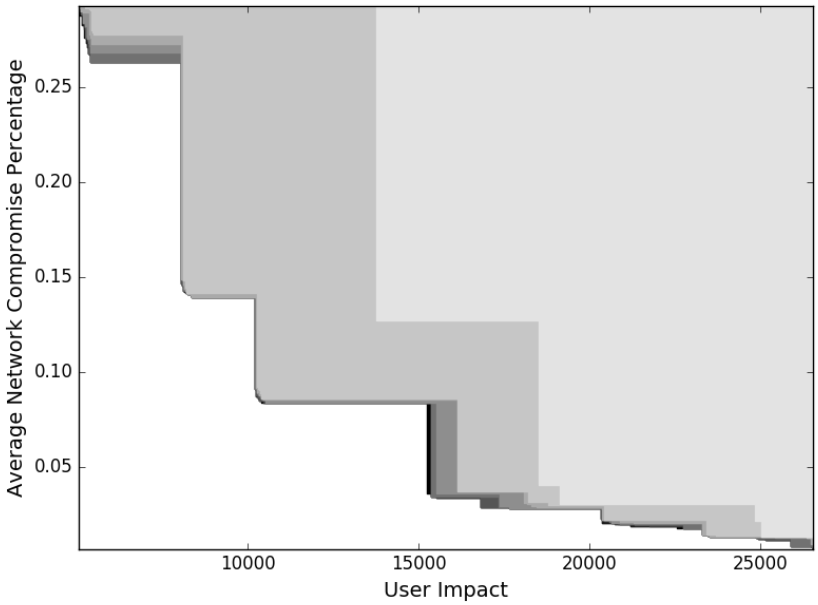
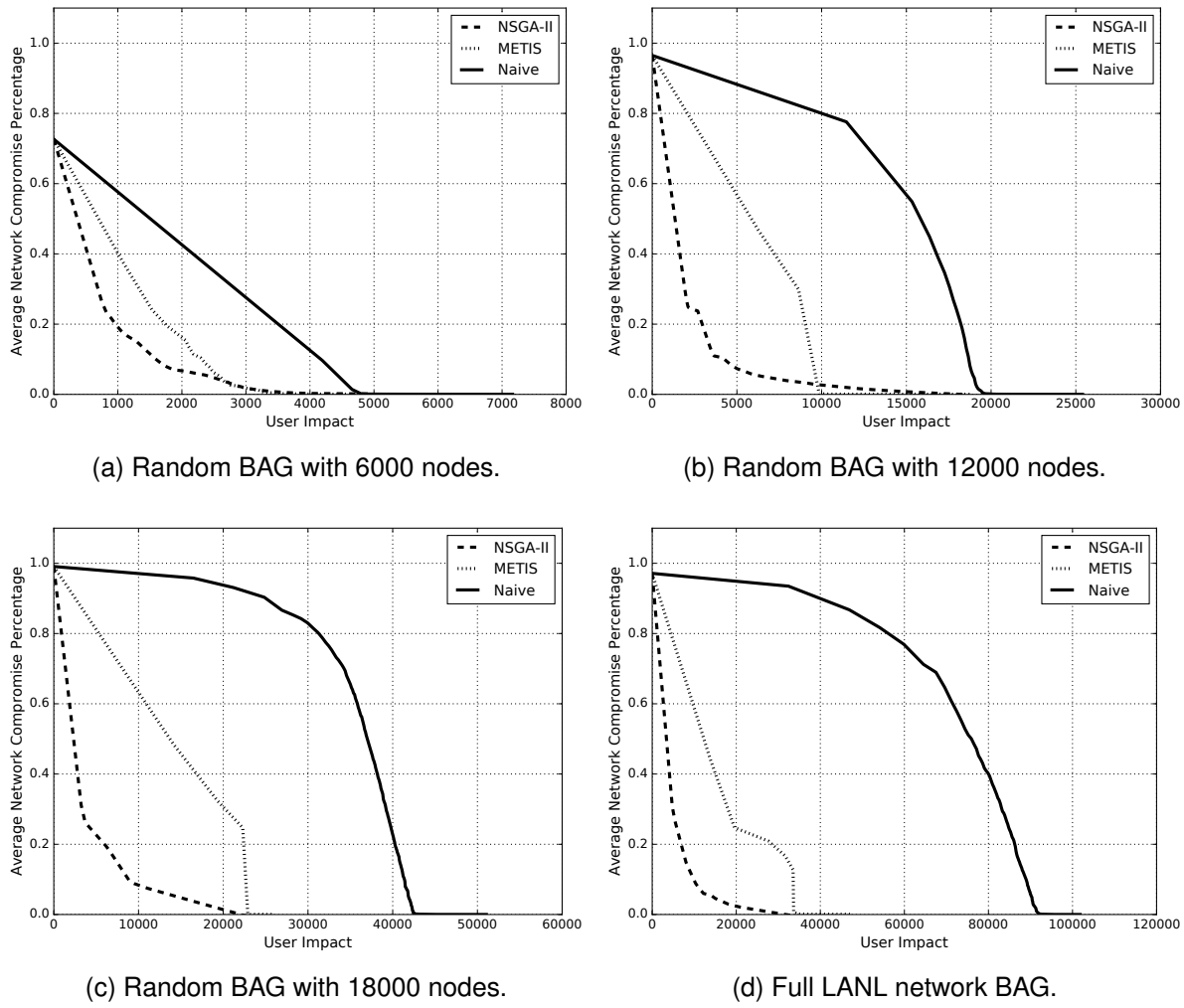


Figure 2.10: Example Pareto front growth during evolution for NSGA-II. The Pareto front is plotted at every 10 generations. The shaded region above each line is the search space area dominated by the corresponding Pareto front, with lighter areas corresponding to earlier generations.

The weighted regression lines of the NSGA-II results for various input BAGs are compared to those achieved by both the naive approach and METIS' k-way partitioning in Figure 2.11. Figure 2.12 compares the required execution time of each method, including a brute-force search for the Pareto optimal set of partitions for sufficiently small BAGs. The comparisons of the objective values achieved for these small BAGs are omitted, because both the NSGA-II and METIS approaches are able to find optimal partitions for these trivial applications. Although the NSGA-II approach produces consistently superior partition solutions for the non-trivial BAGs, the method also takes significantly longer to converge than the less-informed methods, especially as the size of the BAG grows. Figure 2.13 shows a comparison of the objective values achieved by each method when applied to the randomly weighted LANL network BAG.

A statistical comparison method for multi-objective optimizers described by Knowles and Corne is used to compare the results of the NSGA-II approach against the naive and METIS

methods [45]. This evaluation method determines the portion of the trade-off curve each approach statistically outperforms the alternative. Table 2.3 shows the pairwise comparison results of this process.



(a) Random BAG with 6000 nodes.

(b) Random BAG with 12000 nodes.

(c) Random BAG with 18000 nodes.

(d) Full LANL network BAG.

Figure 2.11: Objective value comparison of solutions from the naive approach (Section 2.5.1), METIS k-way partitioning (Section 2.5.2) and NSGA-II method (Section 2.5.3).

2.8 Discussion

Despite the fact that the naive approach is unrestricted in its ability to completely disconnect user nodes from the BAG, it is not surprising to see that it performs so poorly. The method seeks to remove the maximum number of edges at each iteration, leading to extremely high levels of user impact with no regard for the improvement to security. However, it provides a very simple method to use as a baseline for comparison. Because the METIS approach relies

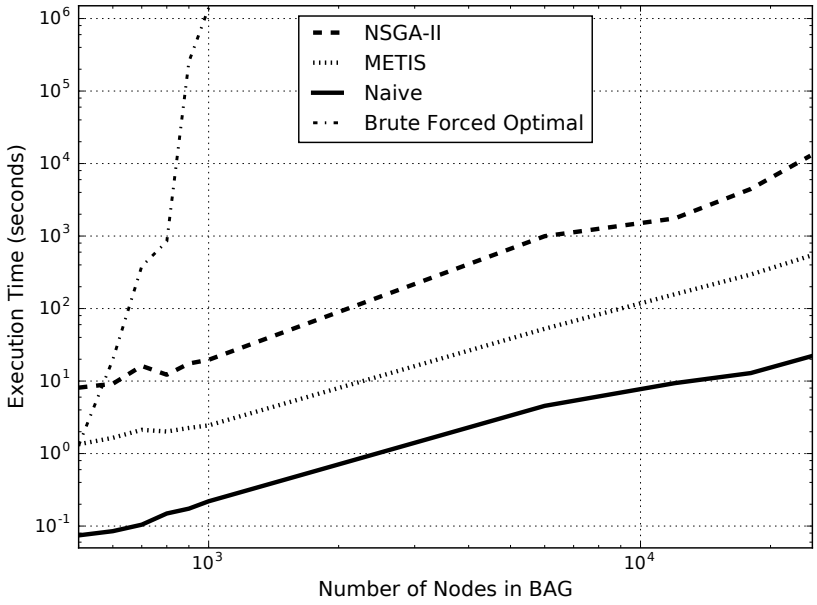


Figure 2.12: The execution time in seconds required for each method to partition BAGs of various sizes.

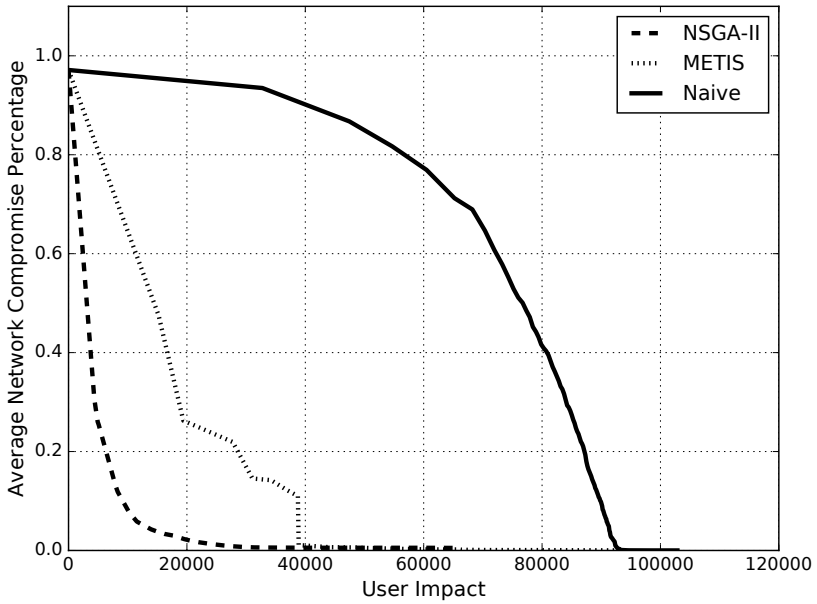


Figure 2.13: Objective value comparison of solutions from the partition methods applied to a randomly weighted version of the full LANL network BAG.

Table 2.3: Objective value comparison between NSGA-II and the less informed methods. The results were gathered from a sampling of 10000 randomly selected goal vectors drawn across a solution set from one run of each method. The Superior Percentage value next to each method shows the percentage of these sample lines for which that method is statistically superior to the other with 95% confidence.

Target BAG		**Superior Percentage**		
Random BAG	NSGA-II	91.91%	4.24%	Naive
6000 nodes	NSGA-II	95.27%	2.72%	METIS
Random BAG	NSGA-II	94.98%	2.48%	Naive
12000 nodes	NSGA-II	96.43%	2.02%	METIS
Random BAG	NSGA-II	94.89%	0.69%	Naive
18000 nodes	NSGA-II	95.59%	0.29%	METIS
LANL BAG	NSGA-II	95.85%	0.90%	Naive
Unweighted	NSGA-II	96.29%	0.36%	METIS
LANL BAG	NSGA-II	96.50%	0.60%	Naive
Weighted	NSGA-II	96.80%	0.24%	METIS

on a low-cost partition heuristic, it significantly outperforms the naive method at all levels of user impact. As the number of nodes in the BAG increases, this performance gap continues to widen.

The NSGA-II approach, with the additional capability to perform user node splits, further improves over the METIS method. The results in Table 2.2 show several solutions that reduce the Average Network Compromise Percentage (ANCP) by an order of magnitude without a single edge removal. The majority of the evolved solutions have less than a hundred edge removals, which suggests that the only edges removed are the result of the cap on the number of times a user node can be split. A transition is apparent at a User Impact level of 14,962, where the cost of edges removed increases dramatically as the NSGA-II approach begins to rely more heavily on METIS to supplement user node splits with additional edge removals. The effect of this combination is dramatic, and further edge removals lead to the ANCP decreasing by yet another order of magnitude.

It is interesting to note that although there are many solutions with a high number of user splits and few edge removals, the opposite case is not present. This highlights the limitations of partitioning methods that are restricted to edge removals. User node splits, when they can be

implemented, are a far more effective method of disconnecting the large connected components of a BAG.

Table 2.3 shows that regardless of the number of nodes in the BAG, NSGA-II is statistically superior to both of the less-informed methods for the vast majority of the trade-off space between User Impact and ANCP. The inclusion of weighted edges also has no apparent effect on the performance improvement of the NSGA-II approach. This is encouraging, since a practical application is likely to involve some authentication edges that would be more difficult to remove. The naive method performs slightly better than the METIS approach in this comparison. This is the result of the region with extremely high User Impact values, which can be seen as the right-most tails of the plots in Figure 2.11. Unlike the METIS and NSGA-II methods, the naive approach is unrestricted in its ability to completely disconnect user nodes from the BAG. However, the fact that the naive method performs so well in this region is of little interest, since it is unlikely that any of these partition solutions could be applied in a practical application.

In exchange for superior partition solution quality, the NSGA-II approach requires significantly more execution time, as can be seen in Figure 2.12. The steep increase in the execution time of the brute-force search is a result of the combinatorial explosion in the size of the search space as the number of nodes in the BAG increases. NSGA-II mitigates this cost increase, but still typically requires more than an order of magnitude increase in computation time compared to the less-informed methods. NSGA-II converges within a few hours for the entire LANL network BAG, which is likely to be insignificant compared to the time required to implement the security configuration changes recommended by the evolved partition solutions, especially if this implementation is done manually.

2.9 Conclusion

Credential theft attacks pose a serious security risk to large enterprise networks, especially those utilizing centralized authentication systems. Modeling a computer network as a BAG makes it easy to identify the potential damage of such attacks. Traditional graph partitioning methods can be used on these graphs to suggest security configuration changes that restrict the ability of adversaries to compromise large portions of the network. However, these methods do

not leverage the problem specific knowledge of the nature of user authentication. Our approach exploits this knowledge and utilizes the strengths of multi-objective evolutionary optimization to produce a collection of dramatically superior solutions, which at various levels of user impact, significantly reduce potential damage of a credential theft attack on the network. This presents the end user with a choice of solutions that do not excessively restrict users, while still minimizing the network vulnerability.

Chapter 3

Evolving Random Graph Generators: A Case for Increased Algorithmic Primitive Granularity

Random graph generation techniques provide an invaluable tool for studying graph related concepts. Unfortunately, traditional random graph models Evolving Multi-level Graph Partition Algorithmstend to produce artificial representations of real-world phenomenon. Manually developing customized random graph models for every application would require an unreasonable amount of time and effort. In this work, a platform is developed to automate the production of random graph generators that are tailored to specific applications. Elements of existing random graph generation techniques are used to create a set of graph-based primitive operations. A hyper-heuristic approach is employed that uses genetic programming to automatically construct random graph generators from this set of operations. This work improves upon similar research by increasing the level of algorithmic sophistication possible with evolved solutions, allowing more accurate modeling of subtle graph characteristics. The versatility of this approach is tested against existing methods and experimental results demonstrate the potential to outperform conventional and state of the art techniques for specific applications.

3.1 Introduction

Graphs are a powerful tool for modeling a wide variety of concepts. Social, computer, transportation or communication networks are common examples. Others include infrastructure applications such as power or water distribution systems. The transmission patterns of contagious diseases are also commonly modeled using graphs. Because these concepts translate so well to graphs, many application specific algorithms are designed to work directly with the

graph representations. For example, computer networks use graph theory to avoid problematic cycles in traffic routing [3].

When new graph algorithms are developed, they typically need to be tested on a variety of graphs to demonstrate versatility and scalability. For some applications, information is readily available to create graphs which model real-world data, such as actual computer networks. In other application domains, this data is in limited supply. For example, deploying wireless sensors to build a graph model can be infeasibly expensive. In these situations, researchers have commonly turned to random graph generation to test their graph algorithms.

However, not all random graph generation techniques are suitable for all applications. Certain types of random graphs are better at naturally representing specific concepts. For instance, wireless sensor network deployment is typically modeled with random geometric graphs, because this random graph model captures the physical proximity requirement needed for short-range communication. When generating random graphs for testing the performance of a new graph algorithm, it is important to select an appropriate random graph model. Whether or not a partitioning algorithm can find high-quality partitions of random geometric graphs is of little importance if the algorithm is intended to be applied to enterprise computer networks, for example.

The selection of a random graph model for a specific application is typically done by comparing a few characteristics of the graph, such as degree distribution or edge density. These coarse selection methods have the potential to miss some of the more subtle characteristics of the concepts to be modeled. For instance, a preferential attachment random graph model produces the power-law degree distribution observed in social networks, but the process might not capture the common presence of certain community structures. When less obvious graph characteristics are not considered, the random graphs produced are likely to be artificial representations of the actual concept. For this reason, random graph generators that are specifically tailored to certain applications can improve the accuracy and appropriateness of these graphs as conceptual models.

Manually developing an application specific random graph model is a complicated process, even when the model only needs to capture a single characteristic, such as an arbitrary

degree distribution [46]. Hyper-heuristics employing genetic programming (GP) have been used in the past to automate the process of developing novel algorithms that are customized to an application [7]. This work leverages the power of GP to create new random graph generation algorithms that are capable of capturing the more subtle graph characteristics often missed by traditional techniques.

3.2 Background

Random graphs and their applications have been studied extensively in previous research. Traditional random graphs are usually described in terms of the mathematical model used to generate them; two of the most common random graph models are Erdös-Rényi and Barabási-Albert.

3.2.1 Erdös-Rényi Random Graph Model

The Erdös-Rényi random graph model, usually referred to as $G(n, p)$, is one of the most basic models, but also one of the most studied [47, 48]. Each of the possible $\binom{n}{2}$ edges is included in the graph with probability p. See Algorithm 6 for an implementation of the Erdös-Rényi random graph model, and Figure 3.1 for an example graph it generated.

Algorithm 6 Erdös-Rényi random graph

```
1: procedure G(n, p)
2:     G ← newGraph()
3:     for i ∈ 1 ... n do
4:         u ← newVertex()
5:         for v ∈ G.vertices do
6:             if random() < p then
7:                 G.addEdge(u, v)
8:         G.addVertex(u)
9:     return G
```

This simple random graph model has proven useful for demonstrating a variety of graph theoretic properties [49]. Unfortunately, it has also been shown to poorly represent real-world systems such as computer networks due to the vertex degree values produced, which follow a Poisson distribution [50, 51].

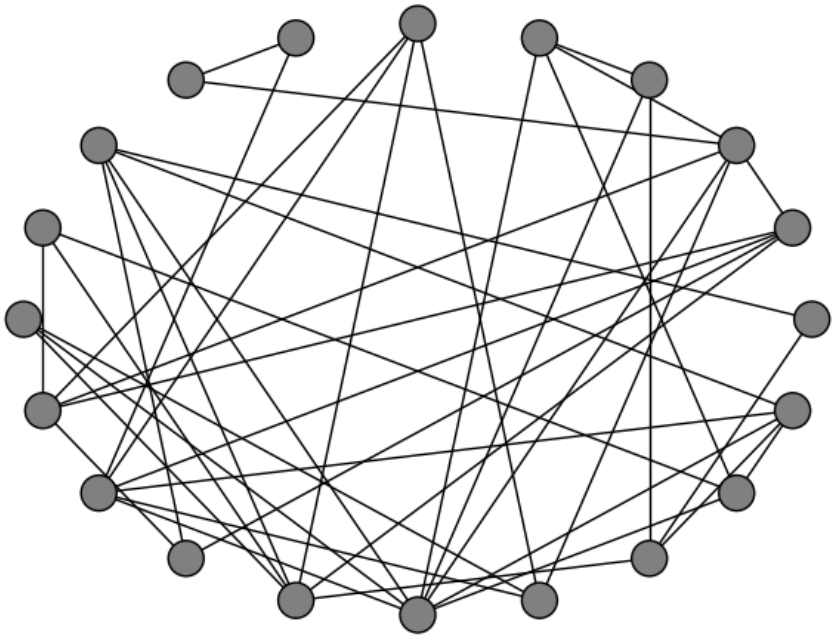


Figure 3.1: Erdös-Rényi random graph for $n = 20$ with $p = 0.2$.

3.2.2 Barabási-Albert Random Graph Model

The Barabási-Albert model [52] improves upon the unrealistic degree distribution of the Erdös-Rényi model. Instead of using a constant probability for including each edge, each new vertex is connected to c existing vertices that are chosen with probability proportional to their degree. As a result, high-degree vertices are connected to more often. This phenomenon is referred to as *cumulative advantage* or *preferential attachment* and produces a power-law degree distribution that is common in graphs which model real-world networks [53, 54]. The Barabási-Albert model is implemented in Algorithm 7 and Figure 3.2 shows a random graph created using the Barabási-Albert model.

Algorithm 7 Barabási-Albert random graph

```
1: procedure G(n, c)
2:     G ← newGraph()
3:     for i ∈ 1...n do
4:         u ← newVertex()
5:         for j ∈ 1...c do
6:             v ← randomVertexByDegree(G)
7:             G.addEdge(u, v)
8:         G.addVertex(u)
9:     return G
```

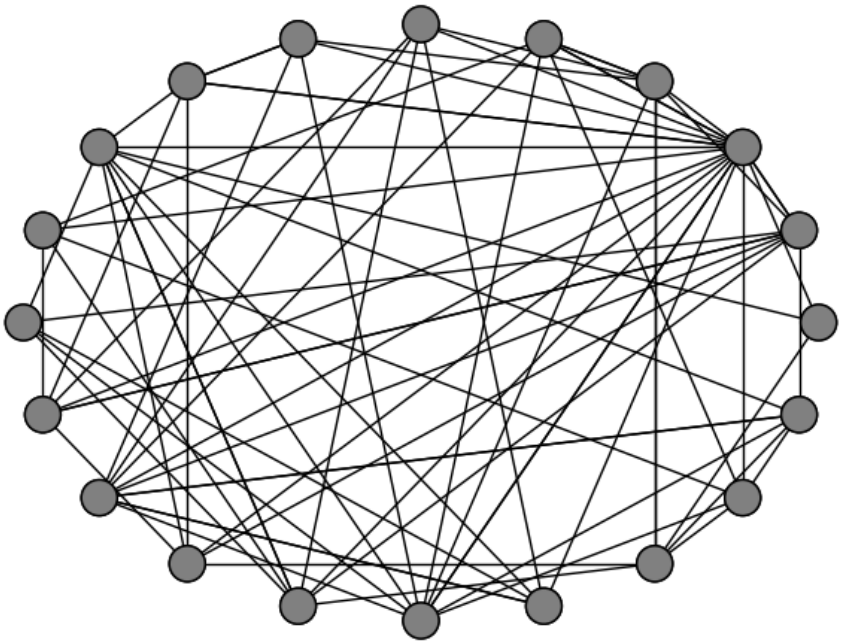


Figure 3.2: Barabási-Albert random graph with $n = 20$ and $c = 2$.

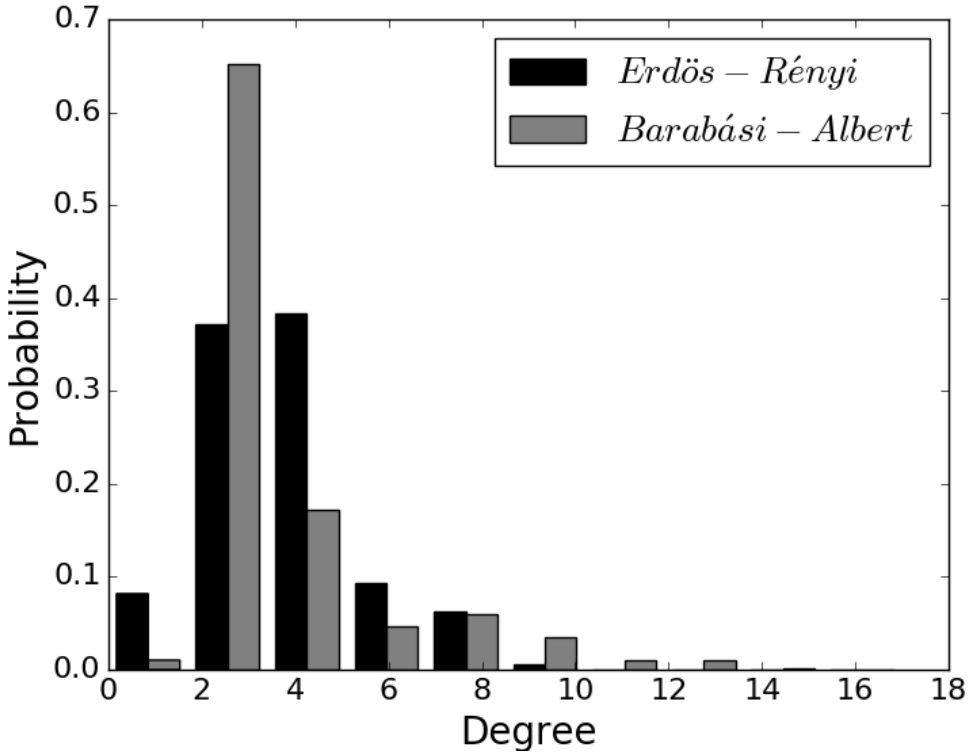


Figure 3.3: Comparison of degree distributions for Erdös-Rényi ($n = 20$, $p = 0.2$) and Barabási-Albert ($n = 20$, $c = 2$) random graph models.

Figure 3.3 compares the degree distribution of Erdös-Rényi and Barabási-Albert random graphs. Although graphs produced by the Barabási-Albert model are similar to real-world networks in terms of their degree distribution, the process still resembles Erdös-Rényi in that it places no limitations on which pairs of vertices can be connected. Many real-world networks have restrictions on connections that cannot be captured by the Erdös-Rényi and Barabási-Albert models. For instance, social networks are often modeled using graphs where vertices represent people and edges correspond to a relationship between two people. Although it is possible for any two people to form a relationship (especially in the case of online social networks), it is more likely for relationships to form between individuals if they are physically nearby, or share mutual contacts. Unfortunately, traditional random graph models lack the complexity needed to reflect such considerations.

3.3 Related Work

Instead of producing a single random graph with the desired properties, this research aims to provide random graph generation heuristics. One possible solution would be to employ heuristic selection techniques. Machine learning has been used to automate the process of selecting the best heuristic for a problem from a set of available heuristics with high accuracy [6]. Unfortunately, this approach is limited by the quality and variety of the set of predefined heuristics; an optimal solution to a given problem cannot be selected if it is not already present in the heuristic set. Techniques that are capable of generating entirely new heuristics help avoid this limitation.

A heuristic that searches to find or create new heuristics is known as a *hyper-heuristic* [55, 7]. Unlike *metaheuristics* [56], which search within the space of possible problem solutions, hyper-heuristics search within the space of possible problem heuristics. This means that instead of searching for a direct solution to a specific problem, hyper-heuristics can select, create, or adapt a heuristic that efficiently finds a solution to the specified problem.

Hyper-heuristics most commonly employ *genetic programming* (GP) to search a problem-specific space of algorithmic primitives. GP is a field of evolutionary computation, which uses a biologically inspired process to evolve a population of solutions to a given problem. In GP,

the solutions being evolved take the form of programs or heuristics. One common method of representing these program solutions is through the use of parse trees [57].

Previous work has demonstrated the potential for GP to evolve custom random graph generation algorithms. Bailey et al. evolved new random graph generation algorithms that mimic the output of traditional random graph models [58]. Harrison implemented a similar approach and studied the use of various graph similarity metrics during solution evaluation [59]. Both of these works assume a common structure for the random graph generator solutions, which can be seen in Algorithm 8. Three components of this process are controlled by the evolved parse tree solutions. Graph initialization (line 3) determines if the graph is initially empty, or contains some basic topology, such as a ring. Inside the main loop body, the graph is "grown" by adding new vertices one at a time. The edge addition step (line 6) determines which vertices, if any, a new vertex is connected to as it is added. During finalization (line 8), existing edges can be removed or rewired at random.

Algorithm 8 Basic structure of a random graph generator

```
1: procedure RANDOMGRAPH(n)
2:     G ← newGraph()
3:     initializeGraph(G)
4:     for i ∈ 1...n do    // "grow" loop
5:         u ← newVertex()
6:         addEdges(u)
7:         G.addVertex(u)
8:     finalizeGraph(G)
9:     return G
```

This common structure is obviously inspired by traditional random graph generation techniques, such as those discussed in Section 3.2. This representation lends itself well to reproducing traditional models, as well as new models that are similar in structure. However, this restriction on the structure limits the search space of possible random graph generating algorithms. For that reason, evolved solutions suffer some of the same drawbacks as the traditional models when attempting to accurately simulate certain types of networks.

This work aims to improve on previous research by relaxing the restrictions on the basic algorithm structure. A more expansive set of operations are made available to the GP when

constructing graph generation algorithms. The new operation set breaks down some of the constructs used in previous work into lower level functionality. For example, the basic "grow" loop is replaced by more general *for* and *while* loop operators, as well as basic *if* and *if/else* conditionals. A larger set of primitive operations will increase the search space of possible algorithm solutions. Previous work has demonstrated that using a larger set of primitive operations can increase the evolution time required to reach convergence, but also improve the overall final solution quality [60].

3.4 Methodology

In order to accommodate a wide variety of possible applications, some of which might involve multiple competing measures of quality, a multi-objective optimization approach is employed. The nondominated sorting genetic algorithm II (NSGA-II) [21], which promotes population diversity without a significant increase in complexity, is used to evolve a population of random graph generating algorithms.

Representation: Solution algorithms are represented using strongly-typed GP parse trees [61]. While it has been demonstrated that the choice of representation can impact the overall performance of the GP [62], the choice of representation was made to isolate the effect of changing the primitive operation set when comparing against previous work.

Initialization: An initial population of parse tree solutions is randomly constructed from the available input and operation nodes. A configurable maximum height parameter is used to limit the size of the initial parse tree solutions. Ramped half-and-half solution generation is used, which produces full parse trees of maximum height for half the population and variable height trees (up to the maximum) for the remainder.

Evaluation: Solutions are evaluated in terms of multiple objectives. The size of the parse tree of a solution is used as a minimization objective to prevent the trees from growing to impractical sizes during evolution. Other objectives used depend on the application, but typically evaluate some metric of the graphs produced by the solution. For example, one objective could

be how closely the generated graphs match the degree distribution of the graphs the generator is meant to reproduce. Any objective that evaluates the graphs produced by a solution are calculated by generating multiple graphs and taking the average objective value.

Parent Selection: Standard NSGA-II parent selection is used, which consists of binary tournaments that favor solutions in less dominated Pareto fronts. Ties are broken using NSGA-II's solution distance metric to encourage population diversity.

Recombination: Due to the destructive nature of parse tree variation operators, offspring are generated using either recombination or mutation, not both. If a pair of parent solutions are selected for recombination, two offspring are produced using random subtree crossover.

Mutation: If recombination is not selected, an offspring is created by cloning a single parent, then performing random subtree replacement.

Survival Selection: NSGA-II's elitist survival selection is used. This approach selects solutions from the least dominated Pareto fronts. Solution diversity is encouraged by using the distance to other solutions in the objective space to break ties for partial Pareto fronts.

Parameters: The parameters for NSGA-II and GP initialization can be seen in Table 3.1. These values were automatically tuned using a random restart hill climbing search.

Table 3.1: NSGA-II and GP parameter values

Parameter	**Value**
Population size and offspring per generation	400
Iterations per evaluation	10
Minimum initial parse tree height	3
Maximum initial parse tree height	5
Recombination probability	65%
Mutation probability	35%

Primitive Operation Set: Solution individuals are constructed from the set of terminal and operation nodes shown in Table 3.2. Except where individually noted, all operation nodes have at least one child operation node, allowing for variable length sequences of operations. The available values for *prob_from_integer*, *integer_constant* and *prob_constant* were chosen to be able to recreate or expand upon the functionality of previous work [58].

Table 3.2: Primitive operation set

Operation Name	**Description**
root	Initializes empty graph, executes child operations, returns final graph
for_index_range	Executes subtree a number of times equal to an integer input value
for_node/edge_loop	Executes subtree once for each node (vertex) or edge in input list
do_while_loop	Executes subtree repeatedly until input conditional is false
if(_then)	Branching based on an input conditional
noop	"no-op", terminates sequence of operations
create_ring/clique/star	Add edges incident to an input list of nodes to create a ring, clique, or star topology
connect_to_nodes_with_prob	As create_star, but add edges according to an input probability
add_edges_with_prob	As create_clique, but add edges according to an input probability
remove/rewire_edges(_with_prob)	Removes or rewires input edges from the graph (optionally according to an input probability)
add_pairwise_edges(_with_prob)	Add pairwise edges connecting two input node lists (optionally according to an input probability)
create_triangles(_with_prob)	As add_pairwise_edges, but for triplets of nodes taken from three input lists
add_stub	Add node to a queue of nodes awaiting edges
connect_stub	Pops node from queue of nodes awaiting edges, connects to another input node
get_all_nodes/edges	List of all nodes or edges
get_incident_nodes/edges	Nodes incident to an input list of edges, or edges incident to an input list of nodes
get_internal_edges	Returns the list of edges whose endpoints are both within an input list of nodes
list_intersection/union	Intersection or union of two lists
list_filter_with_prob	Randomly filters a list according to an input probability
list_portion	First $\lfloor l * p \rfloor$ elements of the input list of length l for probability p
list_shuffle	Returns randomly re-ordered input list
sort_nodes/edges_by_map	Sorts a list of nodes or edges using a $(node : value)$ or $(edge : value)$ mapping
node_degree/betweenness/ closeness_map	Returns a mapping of node centrality values for an input node list
edge_degree/betweenness/ closeness_map	Returns a mapping of centrality values for the incident nodes of an input edge list
average/max_degree	Current average or maximum degree
node/edge_count	Current number of nodes or edges present
true/false_constant	Constant boolean terminal
true_with_prob	True or false according to a probability
bool_and/or	Logical conjunction or disjunction of inputs
less_than	True if the first input numeric is less than the second input, false otherwise
math_add/subtract/ multiply/divide/modulus	Standard math operations (note: division by zero instead divides by 10^{-10})
prob_add/subtract/ multiply/divide/modulus	Same as previous, but clamps output to $[0, 1]$
prob_from_integer	Returns probability from $[0.01, 0.02, 0.025, 0.05, 0.1, 0.2, 0.3, \ldots, 1.0]$ selected using input integer
prob_from_float	Floating point input clamped to $[0, 1]$
integer_constant	Constant chosen randomly from $\{0, 1, 2, \ldots 9\}$
prob_constant	Constant chosen randomly from $\{0.001, 0.01, 0.02, 0.025, 0.05, 0.1, 0.2, 0.3, \ldots, 1.0\}$

3.5 Experiment

The flexibility of the implementation is tested by evolving random graph generators for two example applications. The first application tests the ability of the GP to evolve algorithms which mimic traditional random graph generation techniques. Another application targets a random graph process that generates identifiable communities of well connected subgraphs.

3.5.1 Traditional Random Graph Models

Random graph generator solutions are evolved to recreate the behavior of two traditional techniques: Erdös-Rényi ($n = 100$, $p = 0.05$) and Barabási-Albert ($n = 100$, $c = 2$). The model parameter values were selected to produce small, sparse graphs, since a large number of these graphs will need to be generated throughout the course of evolution. Solutions are evaluated by how similar the graphs they produce are to graphs generated by the target method. In [59], Harrison demonstrated that when evaluating graph similarity for purposes of guiding evolution, there are diminishing returns in terms of solution quality as the number of different metrics used is increased. Comparing the set of degree, betweenness [63] and PageRank [64] centrality distributions was found to strike a balance between evaluation complexity and solution quality. For this reason, these three metrics will be used as competing objectives. For each distribution, the objective value is set to the test statistic returned by a Kolmogorov-Smirnov (KS) test comparing the distributions produced by both methods. This method has been used to demonstrate similarity in distributions before [59], and produces a natural minimization objective as the more similar the distributions, the lower the test statistic will be.

For comparison, a GP developed in previous work is also used to evolve generators targeting this model. See [58] for the implementation details of that approach. Both algorithms are run until convergence is detected by ten consecutive generations with no change to the non-dominated Pareto front, as described in [43]. In order to select a representative solution for comparison, the objective values of the final populations are normalized and summed for each solution. Since all objectives are minimization, the solution with the lowest objective value sum is selected. The final solution chosen from each method is used to generate 100 random

graphs, and the objective values of these graphs are compared using Wilcoxon rank-sum tests at a 95% significance level.

3.5.2 Random Community Graphs

Algorithm 9 describes a process of creating a random graph with k communities. Vertices within the same community are connected with probability p_1. Vertices from different communities are connected with probability p_2. If $p_1 \gg p_2$, edges will be more likely within communities, making them tightly connected. Figure 3.4 shows an example of a graph generated with the random community model using a force based layout. Both random graph GP implementations are run targeting this graph model to determine if evolution can reproduce the underlying community structure.

Algorithm 9 Random community graph generator

```
1:  procedure COMMUNITYGRAPH(n, k, p_1, p_2)
2:      G ← newGraph()
3:      for i ∈ 1...n do
4:          u ← newVertex()
5:          G.addVertex(u)
6:      for i ∈ 1...n do
7:          u ← getVertex(i)
8:          for j ∈ i+1...n do
9:              if (i mod k) == (j mod k) then
10:                 if random() < p_1 then
11:                     G.addEdge(i, j)
12:             else
13:                 if random() < p_2 then
14:                     G.addEdge(i, j)
15:     return G
```

3.6 Results

This section shows some representative experimental results and associated statistical tests. The next section discusses their implications.

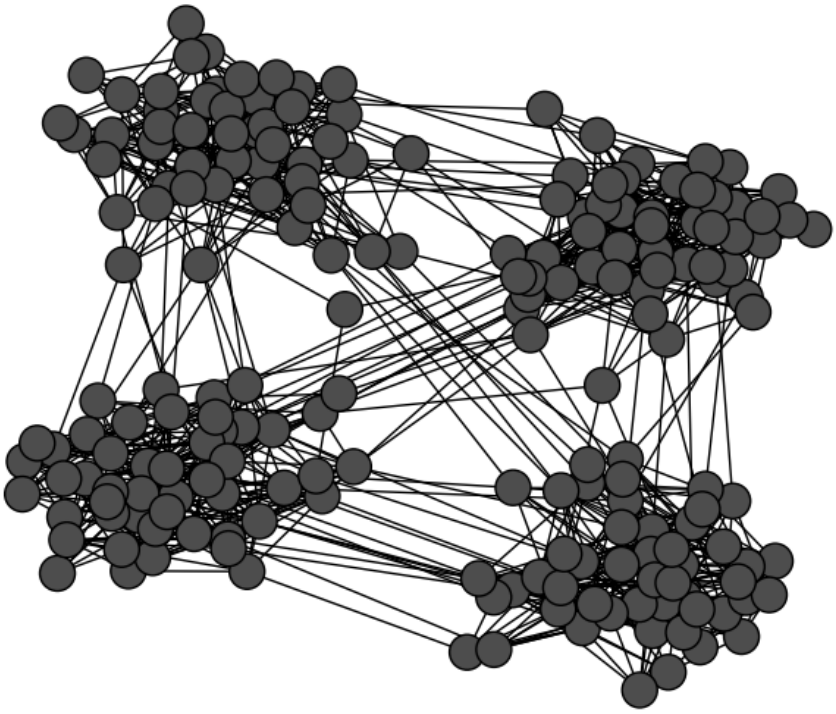


Figure 3.4: Graph generated using Algorithm 9 with $n = 200$, $k = 4$, $p_1 = 0.2$, and $p_2 = 0.005$.

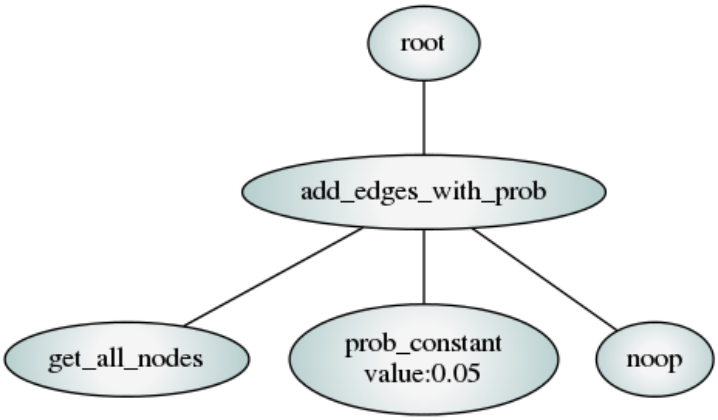
3.6.1 Reproducing Erdös-Rényi

Figures 3.5a and 3.5b show the parse trees of the best solutions produced by the low-level and high-level GP approaches, respectively. Figure 3.6 shows the centrality distribution comparisons for graphs produced using the Erdös-Rényi random graph model (Actual), the high-level GP from previous research (High-GP), and the low-level GP implemented in this work (Low-GP). In each case, both methods are able to closely mimic the required distribution.

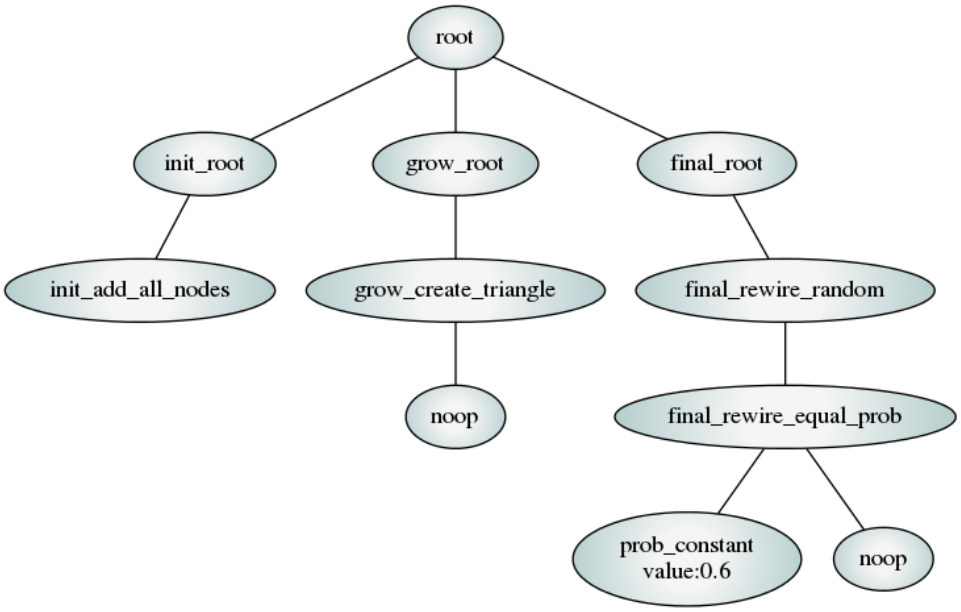
The results of the statistical comparison are shown in Table 3.3, with '$<$', '$=$', and '$>$' indicating better, equivalent, and worse performance, respectively. For all three objectives, the performance difference between the solutions produced by each GP method is statistically insignificant.

Table 3.3: ER objective value comparison

	Low-GP			**High-GP**	
Metric	**Mean**	σ	**Comparison**	**Mean**	σ
Degree	0.101	0.048	$=$	0.108	0.047
Betweenness	0.104	0.031	$=$	0.105	0.033
PageRank	0.110	0.032	$=$	0.112	0.029

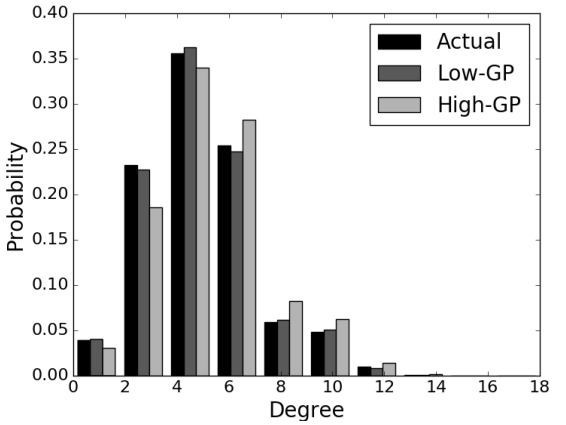


(a) Low-level GP

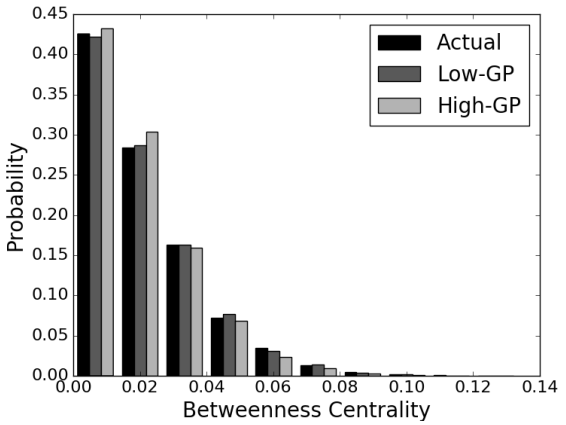


(b) High-level GP

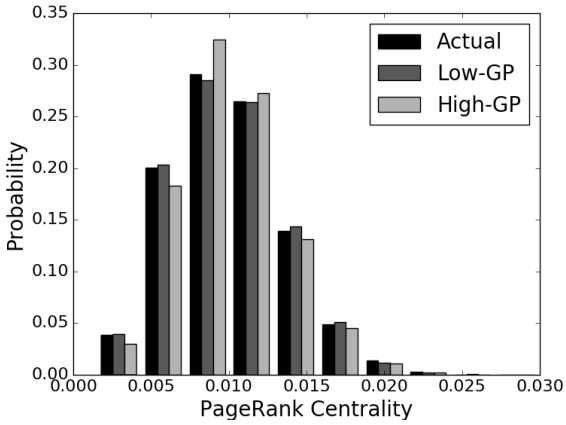
Figure 3.5: Random graph generators produced by both GP approaches when targeted to reproduce the Erdös-Rényi random graph model.



(a) Degree



(b) Betweenness



(c) PageRank

Figure 3.6: Comparison of centrality distributions for Erdös-Rényi random graph model as well as two evolved graph generators.

3.6.2 Reproducing Barabási-Albert

For brevity, the parse trees and distribution comparisons produced for the Barabási-Albert random graph model application are omitted, but the statistical comparison of the objective values achieved for the two GP implementations is shown in Table 3.4. These results indicate that the low-level GP statistically outperforms the high-level GP in terms of PageRank distribution.

Table 3.4: BA objective value comparison

	Low-GP			**High-GP**	
Metric	**Mean**	σ	**Comparison**	**Mean**	σ
Degree	0.058	0.025	$=$	0.060	0.021
Betweenness	0.127	0.049	$=$	0.127	0.043
PageRank	0.112	0.037	$<$	0.130	0.044

3.6.3 Reproducing Random Community

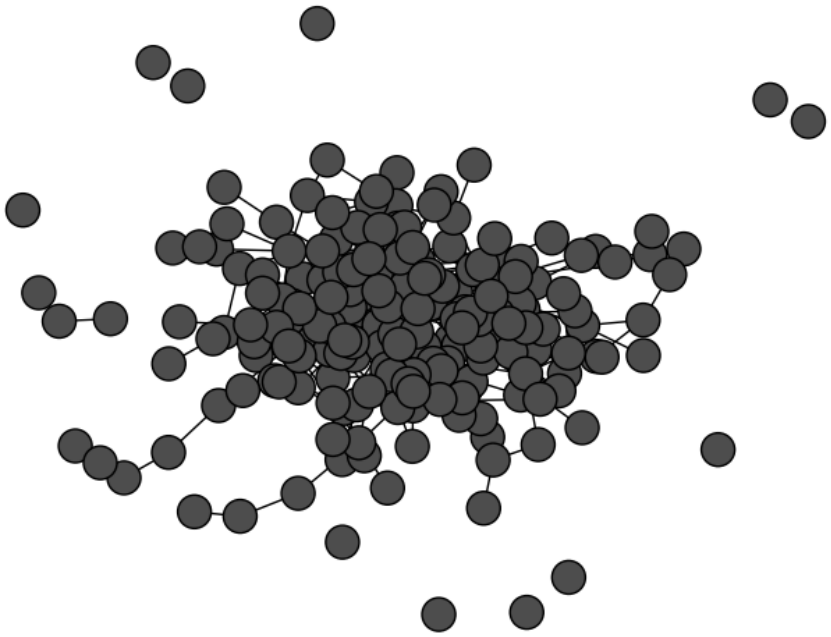
Table 3.5 indicates that the solution produced by the low-level GP statistically outperforms the solution produced by the high-level GP in terms of all three objective values. The reason for this discrepancy in performance is obvious when examining sample graphs produced by each solution. Figure 3.7a shows a graph produced by the high-level GP solution, while Figure 3.7b shows one created by the low-level GP solution. The low-level implementation clearly does a better job of capturing the community structures present in the original model.

Table 3.5: Random community objective value comparison

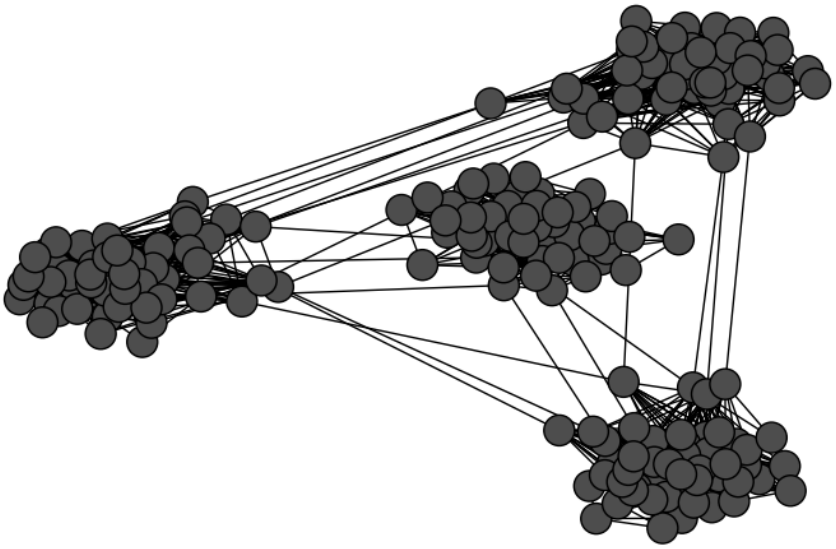
	Low-GP			**High-GP**	
Metric	**Mean**	σ	**Comparison**	**Mean**	σ
Degree	0.436	0.075	$<$	0.458	0.055
Betweenness	0.209	0.105	$<$	0.320	0.126
PageRank	0.127	0.029	$<$	0.150	0.036

3.7 Discussion

Not only are both GP implementations able to almost perfectly reproduce the Erdös-Rényi (ER) graph model, but they both converge quickly on a good set of evolved solutions. This does not



(a) High-level GP



(b) Low-level GP

Figure 3.7: Graphs generated by both GP solutions trained on random community graphs.

come as much of a surprise, however, because the ER model is the simplest of the three applications considered. Although this certainly provides a proof-of-concept, the particular result is not likely to be of much use considering how few real-world applications can be accurately represented using the ER model.

Both GP approaches are able to recreate the behavior of the Barabási-Albert model reasonably well; however, the low-level GP solution manages to achieve a more accurate PageRank distribution. Although in about 90% of the experimental runs, the low and high-level GPs need about the same number of fitness evaluations to converge on good solutions, it is worth noting that in the remaining 10% of the cases, the low-level GP requires almost twice as many evaluations to converge. This is evidence of the drawback of increasing the search space with a lower-level implementation. While the low-level representation allows for a wider range of algorithm possibilities, it also increases the difficulty of finding any specific algorithm. However, the required a priori time of the hyper-heuristic is not typically of critical importance, since this time investment is amortized over repeated uses of the evolved solutions.

The random community graph model application, on the other hand, highlights the strengths of the lower-level implementation. The richer primitive operation set is better able to capture the underlying community structure of the model, which is very obvious when comparing graphs produced by the resulting solutions. This is a promising result for applications that require a more accurate model than what can be achieved by simply comparing one or two basic graph metrics, such as degree distribution. The fixed structure of the high-level GP approach limits the information it can consider when deciding how to place edges. It is easy to imagine any number of graph applications where more information is needed when placing edges. For instance, graphs that model power grids need to account for geographic proximity when connecting two devices due to the nature of the physical properties of the object that the edge represents.

3.8 Conclusion

Random graph models provide an invaluable resource in many research domains. Conventionally, a traditional random graph model is selected to produce graphs which represent some

application specific concept. The selection process is usually based on a small set of graph similarity measures, and this process can even be automated using machine learning techniques. Unfortunately, a selected model might only be an accurate representation with respect to a few graph characteristics, leading to artificial graphs. The model selection process also relies on the set of available models; if an accurate model is not present in the selection set, this approach cannot generate a new, high-quality model tailored to the particular application.

The goal of this research was to address this limitation by automating the development of accurate random graph models for new applications. The platform implemented in this work features a richer set of lower-level primitive operations than those that have been used in the past, allowing for more expressive algorithm representation. The increased flexibility makes it is possible to evolve more sophisticated algorithms that can truly capture a wider range of graph characteristics. Experimental results illustrate that the less restricted representation is capable of capturing more subtle details of a random graph model than normally possible with conventional methods. However, this improvement in modeling accuracy can come at the cost of additional evolution time. This trade-off might not be acceptable for some applications, but the approach still has potential when the accuracy of the model is of utmost importance.

Chapter 4

Evolving Multi-level Graph Partitioning Algorithms

Optimal graph partitioning is a foundational problem in computer science, and appears in many different applications. Multi-level graph partitioning is a state-of-the-art method of efficiently approximating high quality graph partitions. In this work, genetic programming techniques are used to evolve new multi-level graph partitioning heuristics that are tailored to specific applications. Results are presented using these evolved partitioners on traditional random graph models as well as a real-world computer network data set. These results demonstrate an improvement in the quality of the partitions produced over current state-of-the-art methods.

4.1 Introduction

The problem of graph partitioning shows up in a wide variety of application domains. Examples include organizing parallel computation workload [65], VLSI layout design [66], image processing [67], and critical infrastructure protection [68], among others. In general, optimal graph partitioning is known to be NP-hard [18]. As a result, time sensitive applications typically rely on heuristics which provide approximate partition solutions.

One of the most commonly used approaches to quickly find high-quality graph partition approximations is multi-level graph partitioning [34]. The general idea behind multi-level partitioning involves producing a smaller graph which is an approximation of the original input graph. A high-quality partition is calculated for this reduced graph and this partition is mapped back to the original graph. This is typically done over several iterations and the quality of the partition is improved at each iteration through a refinement process. Several well-known

graph partition software packages implement multi-level schemes, such as METIS [19], JOSTLE [35], Scotch [36], and DiBaP [37].

Previous work has shown that partition quality can be improved by selecting specialized heuristics for classes of graphs with specific properties. For instance, superior partition heuristics have been found for graphs with power-law degree distributions [69]. If this process was repeated for a wide variety of applications, it might be possible to assemble a good set of tailored graph partition algorithms instead of relying on a general purpose solution. Selection of the appropriate algorithm for the problem at hand could even be automated, a task for which machine learning approaches have been shown to excel [6].

Unfortunately, this relies on the best solution for a problem already being available. The very nature of these custom heuristics means that the performance gains are likely to be limited to the specific class of graphs for which they were developed. To achieve the same improvements for a new application, the process of manual algorithm optimization must be repeated. Alternatively, this process of developing algorithms tailored to a specific application can be automated by searching the space of custom heuristics.

Genetic programming (GP) [57], a field of evolutionary computation, has been shown capable of automatically generating [7] and optimizing heuristics for a variety of applications [70]. Utilizing GP to optimize multi-level partitioning algorithms can provide two distinct advantages. First, the evolutionary process will consider heuristics that might have been overlooked during manual development because they are not intuitive. Second, once the framework for evolving custom heuristics for a specific application is constructed, it can be quickly applied to any number of other problems as the need arises. The work presented in this paper investigates the potential of using GP to automate the process of tailoring multi-level partitioning algorithms for specific applications, improving their performance over more generalized state-of-the-art partition methods.

4.2 Graph Partitioning

Given an integer $k \geq 2$ and a graph $G = (V, E, w_v, w_e)$ with the set of vertices V and the set of edges E, vertex weight vector w_v, and edge weight vector w_e, a k-way graph partition

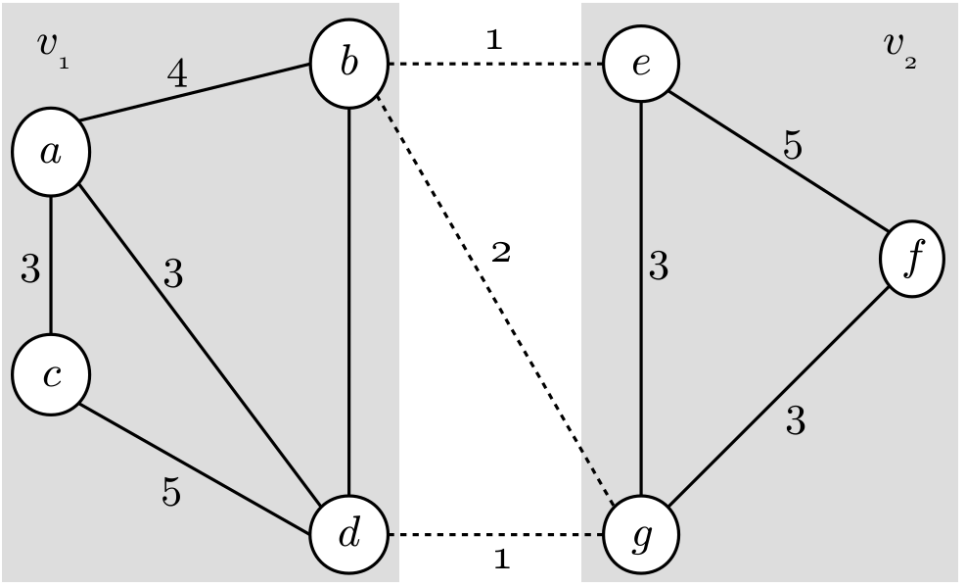


Figure 4.1: Example graph partition with vertex sets $v_1 = \{a, b, c, d\}$ and $v_2 = \{e, f, g\}$. Edges between partitions are indicated by dashed lines. The cut-cost of the partition is 4.

divides the vertices of V into k subsets $V_1, V_2, \ldots, V_k$, such that $V_i \cap V_j = \emptyset$ if $i \neq j$ and $V_1 \cup \cdots \cup V_k = V$. For unweighted graphs, let all the entries in w_v and w_e be one. The total weight for a set of vertices X is given by:

$$W_v(X) = \sum_{i \in X} w_v[i].$$

Similarly, the total weight of a set of edges Y is:

$$W_e(Y) = \sum_{j \in Y} w_e[j].$$

For a given partition, let E' be the subset of edges from E that connect vertices in different partitions. $W_e(E')$ is the weight of this edge set, and is known as the *cost* or *cut-cost* of the partition. Typical graph partition applications require this cost be minimized. See Figure 4.1 for an example graph partition.

Many applications also place restrictions on the relative total weight of the partitioned vertex subsets. A *balanced* partition requires that

$$\frac{\max_{i=1\ldots k} W_v(V_i)}{\sum_{j=1\ldots k} W_v(V_j)/k} \leq 1 + \epsilon$$

for some constant imbalance factor ϵ. In other words, the ratio of the weight of the heaviest partition to the weight of the average partition cannot exceed $1 + \epsilon$.

4.2.1 Multi-level Graph Partitioning

Multi-level graph partitioning is one of the most widely used graph partition approximation methods. The approach generally consists of three distinct phases, typically referred to as the *coarsening*, *partition*, and *uncoarsening/refinement* stages. See Figure 4.2 for a visualization of the multi-level graph partitioning approach.

Coarsening

During the coarsening phase, a smaller approximation of the input graph is created. The coarsening process is repeated, creating a sequence of smaller and coarser graphs, until the size of the coarsest graph is sufficiently small.

The smaller approximation graphs are typically obtained by performing edge or subgraph contractions on the input graph. One common approach to selecting edges for contraction is to find a maximal matching. A maximal matching can be created in a variety of ways, but generally some simple heuristic is used to keep the complexity of the coarsening phase down. Some example heuristics that have been investigated in previous research include:

Random matching: Unmatched vertices are visited in a random order and an incident edge is randomly selected from those that do not violate the matching.

Light edge matching: Similar to random matching, but the lowest weight incident edge is selected instead of selecting randomly.

Heavy edge matching: Identical to light edge matching, except favoring heavy weight edges.

While coarsening using matching schemes has worked well for some applications [19], it has been shown that graphs with power-law degree distribution are difficult to coarsen with matchings alone. In these instances, improved performance can be achieved by contracting small, highly connected subgraphs instead [69].

Partition

During the partition phase, a direct partitioning approach is used to partition the coarsest graph. Due to the small size of the coarsest graph, very little time is required to get a partition of relatively decent quality. For this reason, more computationally expensive partition methods can be employed, such as spectral partitioning [71] or Kernighan-Lin (KL) [72]. Karypis et al. demonstrated that even simpler partition approaches can be used without a loss of final partition quality [19]. Some examples of these simple methods include:

Graph growing partition (GGP): A partition is grown by visiting a random vertex, then adding vertices to the partition in a breadth-first fashion until the partition contains the necessary vertex weight.

Greedy graph growing partition (GGGP): Similar to GGP, but neighboring vertices are added to the partition in an order which maximizes the decrease in the cost of the partition.

Uncoarsening and Refinement

During uncoarsening, the partition solution for the coarsest graph is mapped back to the next coarsest graph. The partition for the coarsest graph gives a good starting partition for the next coarsest, but the quality of the partition is then improved through a refinement step. This uncoarsening and refinement process is repeated until a refined partition is found for the original input graph. Multiple partition refinement strategies exist, and some examples include:

KL refinement: The partition to be refined is used as a starting point for the Kernighan-Lin partition algorithm, except each pass of the algorithm terminates if a configurable number of vertex swaps do not decrease the cost of the partition.

Greedy refinement: The KL refinement algorithm, limited to a single pass.

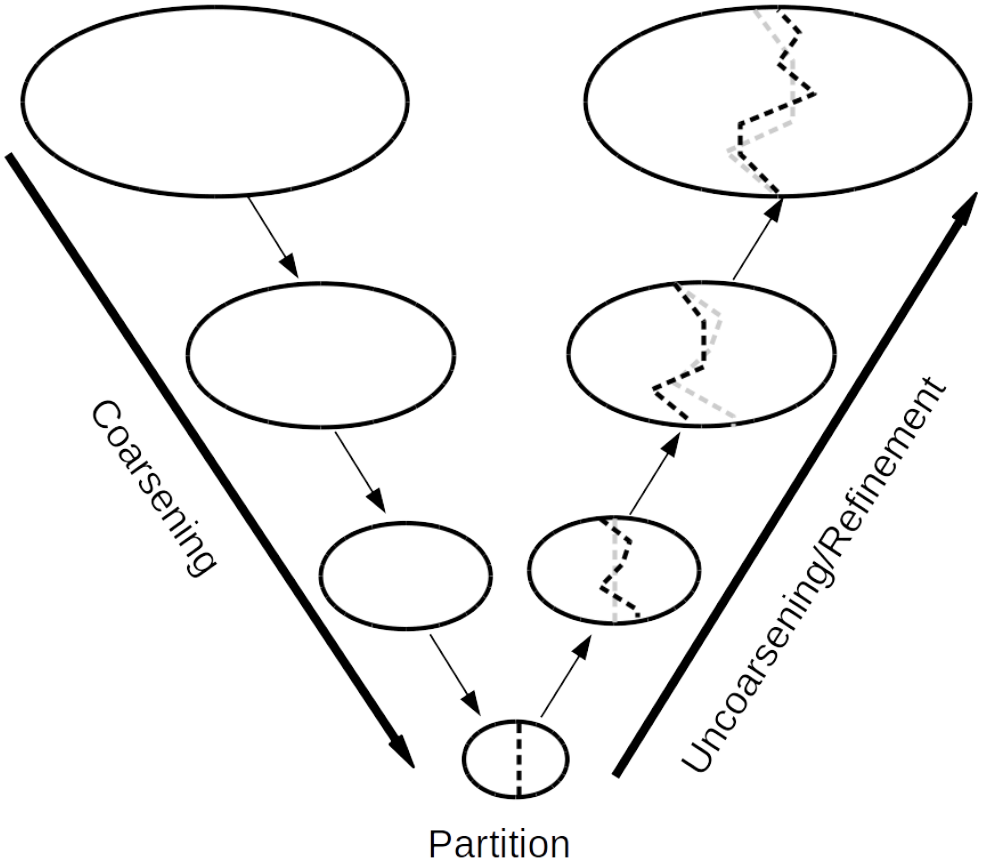


Figure 4.2: Multi-level graph partitioning strategy.

4.3 Evolutionary Computation

Evolutionary algorithms (EA) are a family of biologically inspired generate-and-test black-box search algorithms [20]. This process encourages solutions with higher *fitness* values, which is a measure of the solution quality, or how well it solves the problem at hand. The stages of a typical EA consist of:

Initialization: A population of solutions is randomly generated and evaluated.

Parent selection: Solutions are randomly selected from the population (typically favoring higher fitness) to participate in creating new offspring solutions.

Recombination: Offspring solutions are created using the genetic information from multiple parent solutions.

Mutation: Offspring solutions are stochastically altered to facilitate exploration of the search

space.

Survival selection: The new generation of offspring is evaluated and is either added to, or replaces the current population. A subset of the population is selected to "survive" and continue on in future generations. Again, this selection process usually favors higher fitness.

Termination: The process of selecting parents, creating offspring, and selecting survivors continues until some termination criteria is met. Some example termination criteria are reaching some threshold of quality, convergence of the population, or some limit on total execution time.

4.3.1 Genetic Programming

Genetic programming (GP) is a field of evolutionary computation where the solutions being evolved take the form of programs or algorithms [57]. A set of primitive operations is usually constructed by observing the common and essential elements of algorithms which have been designed to solve the intended problem. This primitive operation set is used as algorithmic building blocks by the GP to piece together new candidate algorithm solutions.

Many forms of representing algorithm solutions have been developed, but one of the oldest and most common approaches represents programs as parse trees. With this representation, offspring are generated using subtree crossover, where a random node is selected in the parse trees of both parents, then the subtrees rooted at these nodes are swapped to generate two offspring. Mutation is accomplished by randomly choosing a node and replacing the subtree rooted at that node with a new, randomly generated tree. See Figure 4.3 for an example parse tree representing a simple partitioning algorithm.

4.4 Related Work

This work was inspired by previous research that investigated the effect of the coarsening scheme used for multi-level partitioning algorithms. Abou-Rjeili et al. developed new heuristics for graph coarsening that improved the partition quality for graphs with power-law degree distributions [69]. The superior performance achieved suggests that there is potential in specializing these algorithms to specific classes of graphs. The framework developed in this research

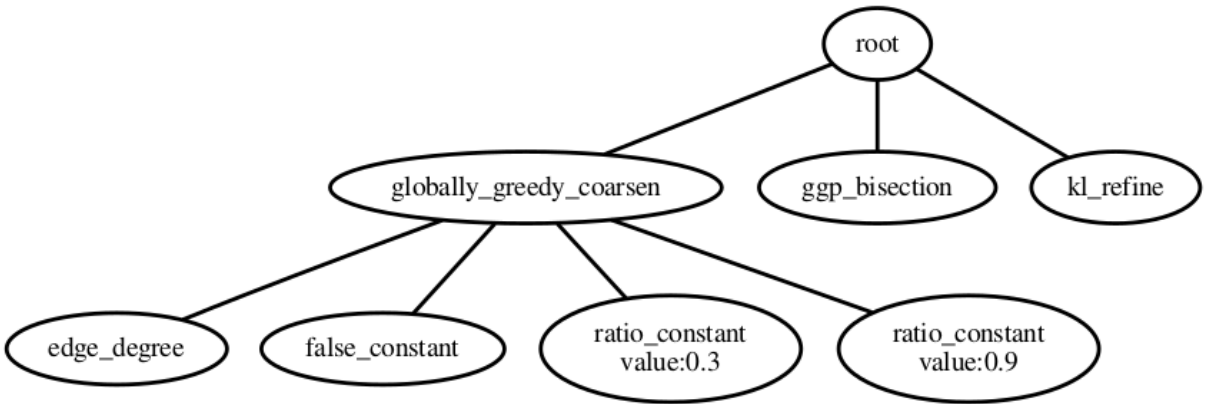


Figure 4.3: Simple genetic programming parse tree example composed of operations described in Section 4.5.1.

will have the added benefit of being able to quickly develop partitioning algorithms which are tailored to new problem areas simply by re-running the GP.

There are many examples of evolutionary computation techniques being used to find approximate minimum graph partitions [38]. The Karlsruhe Fast Flow Partitioner Evolutionary (KaFFPaE) leverages the inherent parallelizability of evolutionary algorithms to evolve graph partitions on a distributed system [40]. Soper et al. introduced an evolutionary search algorithm that makes use of a multilevel heuristic for crossover to generate high quality graph partitions [41]. Benlic et al. developed a multilevel memetic algorithm for the k-way graph partitioning problem [42]. While these approaches are capable of finding very low cost partition solutions, they do so at the cost of execution time. This trade off makes them suitable for applications which must infrequently find extremely high quality partitions, but inappropriate for more time-sensitive problems. This work instead aims to invest a large amount of a priori evolution time to produce algorithms that are capable of quickly finding high quality partition solutions for a specific class of graphs.

The strengths of GP have been leveraged in previous work to evolve random graph generation algorithms [58, 59]. While these works aim to solve a different problem, they still evolve graph related algorithms. Because of this similarity, there is potential overlap in the primitive operation sets used to construct candidate solution algorithms.

4.5 Methodology

Genetic programming is used to evolve a population of mutli-level graph bisection algorithms that minimize the cost of the partitions they produce. Note that this work is limited to bisectioning, but could be extended to more general k-way partitioning through the use of recursive bisectioning applications.

Representation: Algorithm solutions are expressed as parse trees. A strongly typed representation is employed to accommodate the three distinct phases of the partition algorithms [61]. Initial parse trees have a configurable maximum height to begin the search with simple heuristics that can grow during the course of evolution.

Initialization: The population was initialized using a ramped half-and-half method, which produces full parse trees of maximum height for half the population and variable height trees (up to the maximum) for the remainder.

Evaluation: Each candidate solution is used to partition a configurable number of graphs of the relevant type. The solution's fitness score is given by

$$Fitness = \frac{1}{|P|} \sum_{p \in P} \left[\sum_{(u,v) |\ p[u] \neq p[v]} w_e \left[(u, v) \right] \right],$$

where P is the set of partitions produced by the evolved solution and w_e is the vector of edge weights as described in Section 4.2. If a solution algorithm takes an excessive amount of time to compute a partition, or produces partitions that violate the balance constraint described in Section 4.2, a penalized fitness value is assigned, which is given by

$$Penalized\ Fitness = \sum_{(u,v) \in E} w_e \left[(u, v) \right],$$

where E is the complete set of edges in the input graph. In other words, the penalized fitness is the cost of a partition which removes every edge from the graph. Lower fitness values are considered superior, which encourages algorithms that quickly produce low cost partitions while respecting balance.

Table 4.1: GP parameter values

Parameter	Value
Population size and offspring per generation	60
Partitions per evaluation	10
Minimum parse tree depth	2
Maximum parse tree depth	5
Mutation probability	25%
Termination threshold	30

Parent selection: Parents are selected using binary tournaments, which randomly select two solutions from the population, then return the highest fitness of the two. The low tournament size lowers selection pressure to counteract the elitism introduced by survival selection.

Recombination: 95% of the offspring are created using subtree crossover from two donor parent solutions as described in Section 4.3.1.

Mutation: The remaining 5% of the offspring are created by performing subtree replacement mutation on a single donor parent as described in Section 4.3.1. Because subtree crossover and subtree replacement both have the potential to dramatically alter a solution, only one method is applied to each offspring.

Survival selection: Truncation selection is used for survival, simply selecting the fittest individuals. This approach is very elitist, and encourages exploitation of currently known high fitness solutions.

Termination: Evolution is terminated when the best fitness seen has not improved for a configurable number of consecutive generations.

The values for the parameters of the GP can be seen in Table 4.1. These parameters were tuned using a random restart hill climbing search.

4.5.1 Primitive Operation Set

The individuals in the population of the GP are constructed from the following set of operations.

Root Node: All solutions use the same operation for the root node of their parse tree. This node has three child nodes, which correspond to the three phases of the multi-level partition approach. The first child node takes a graph as input and returns a coarsened graph. This process is repeated, storing the sequence of coarsened graphs, until the coarsest graph contains at most fifty vertices. The second child node takes the coarsest graph as input and returns an initial partition assignment of the vertices. Finally, the third child takes two consecutive graphs from the sequence of coarsened graphs, along with a partition assignment, and returns a refined partition assignment for the less coarse graph. The uncoarsening and refining step is repeated, working from the coarsest graph back to the original graph, until the partition assignment for the original input graph is obtained, which is returned as the final result of the algorithm.

Graph Coarsen Nodes: The first set of coarsening nodes are inspired by traditional multi-level partitioning approaches.

Random matching coarsen: Coarsens the input graph by contracting the edges of a random maximal matching.

Heavy edge matching coarsen: Contracts the edges of a heavy edge maximal matching, as described in Section 4.2.1.

Light edge matching coarsen: Contracts the edges of a light edge maximal matching, as described in Section 4.2.1.

The remaining nodes are inspired by the coarsening schemes developed by Abou-Rjeili et al. [69].

Globally greedy coarsen: This node takes input from four child nodes. The first provides a formula which evaluates an edge in the graph and returns a metric value. The second returns a boolean that determines if the preceding metric is to be maximized or minimized. The third returns the *maximum vertex weight ratio*, which is the portion of the entire graph's total vertex weight that an individual contracted vertex cannot exceed. The fourth returns the *maximum contraction ratio*, which determines the percentage of the vertices that can be contracted during a single coarsening phase. This operation sorts all of the edges in the graph using the metric formula and attempts to contract them in order, skipping any edge contraction that would violate the maximum vertex weight restriction. The process terminates when the maximum contraction

ratio is reached, or all edges are considered, whichever occurs first.

Locally greedy, globally random coarsen: Identical to the globally greedy coarsening strategy, except for the procedure used to generate the list of edges for contraction. Instead of ranking the graph's entire set of edges, the list of edges is built by randomly visiting vertices in the graph and using the metric to select one incident edge using the edge metric input.

It is worth noting that Abou-Rjeili et al. fixed the values of the maximum vertex weight ratio and the maximum contraction ratio to 0.05 and 0.5, respectively. This work instead chooses to allow evolution to attempt to optimize the values for these parameters.

Edge Metric Nodes: The following metrics were chosen because they can be calculated without increasing the overall complexity of the multi-level partitioning algorithm. Note that these values can be combined and manipulated using math operations.

Edge degree: Returns the sum of the degrees of the vertices incident to the edge.

Edge weight: The weight of the edge.

Edge node weight: The sum of the weights of the vertices incident to the edge.

Edge core number: The sum of the core numbers of the vertices incident to the edge. For a description of node core numbers, see [73].

Math Operators: Basic addition, subtraction, multiplication, division, modulus, exponentiation, additive and multiplicative inverse. Some of these operators require special attention due to the stochastic nature of the process. For example, if division would produce a division by zero exception, it instead divides by a value very close to zero.

Numerical Constants: These nodes return a constant value that is randomly chosen once during initialization.

Ratio constant: Randomly selected value from $\{0.1, 0.2, \ldots, 1.0\}$.

Probability constant: Randomly selected value from $\{0.001, 0.01, 0.02, 0.025, 0.05, 0.1, 0.2, 0.25, 0.5, 1.0\}$.

The possible values for these nodes were chosen to allow the GP to recreate and expand upon the functionality of existing heuristics.

Boolean Nodes: True and false constant nodes, as well as a node that randomly returns true according to an input probability.

Partition Nodes: These nodes take a graph as input and return an assignment of vertices into partitions.

Random bisection: Randomly assigns vertices into two partitions. The order of the vertices is randomized and then iterated through. Vertices are added to the first partition until the partition exceeds half the total vertex weight. The remaining vertices are assigned to the second partition.

GGP bisection: Graph is partitioned using graph growth partitioning, as described in Section 4.2.1.

GGGP bisection: Graph is partitioned using greedy graph growth partitioning, as described in Section 4.2.1.

Spectral bisection: Spectral partitioning is used to bisect the graph (see [71]).

KL bisection: Graph is bisected using the Kernighan-Lin algorithm (see [72]).

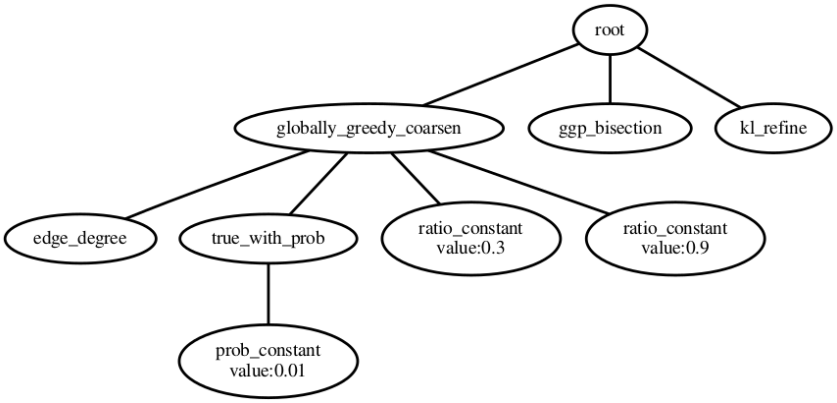
Uncoarsening Nodes: Two uncoarsening nodes are used, which only differ in the refinement method they employ.

KL refinement: Kernighan-Lin refinement, as described in Section 4.2.1.

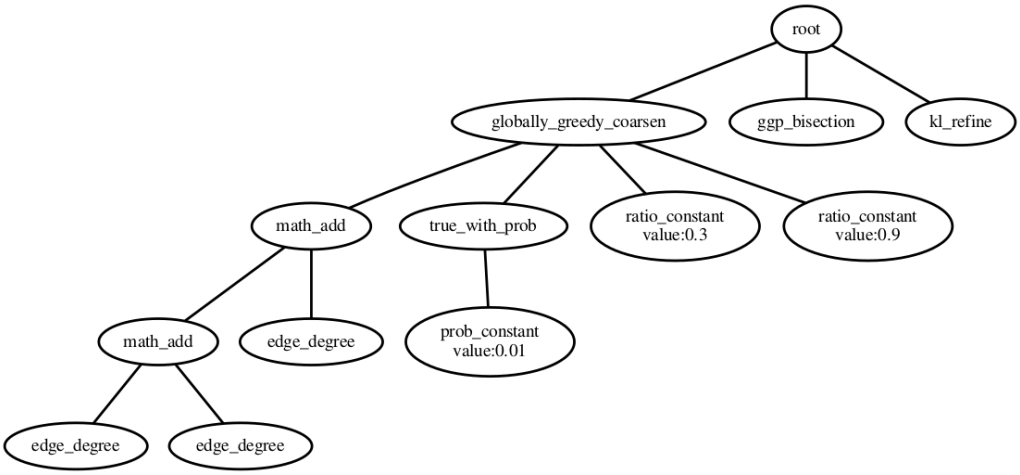
Greedy refinement: Greedy partition refinement, as described in Section 4.2.1.

4.6 Experiment

The GP approach is used to evolve multi-level partition algorithms for three types of graphs. The first two applications are targeted at partitioning graphs from two specific random graph models: Erdös-Rényi [47], and Barabási-Albert [52]. These models, which are known to have different degree distributions, were selected to illustrate the effectiveness of algorithm specialization. In order to demonstrate real-world applicability, the third application targets graphs created from actual network data released by Los Alamos National Laboratory (LANL) [23]. One month of the network data set was modeled as a bipartite graph with 9,924 user vertices, 14,822 computer vertices, and 106,693 authentication edges. Subgraphs were created by inducing the set of vertices visited by a random walk of the total graph.



(a) Erdös-Rényi



(b) Barabási-Albert

Figure 4.4: Example evolved partition algorithms for both random graph model applications.

During solution evaluation, a set of the application specific random graphs are generated, each with 100 vertices. The size of the graphs are kept small because a large number of these graphs will need to be generated during the full course of evolution. The candidate solution being evaluated is used to partition each graph in the set, and the solution's quality is determined by the average cost of the partitions produced. By using multiple randomly generated graphs for each evaluation, evolution encourages solutions which are good at partitioning that class of graphs instead of overspecializing on a small, fixed set of specific graphs.

A separate set of thirty verification graphs are generated to evaluate the performance of partition algorithm solutions from the final population of each GP run. For comparison, the verification graphs were also partitioned using standard spectral partitioning as well as the k-way partitioning function of the METIS software library. To examine the extent to which the

Table 4.2: Relative average partition cost

Method	$\mathbf{E_{ER}}$	$\mathbf{E_{BA}}$	$\mathbf{E_{LANL}}$	**METIS**	**SP**
$\mathbf{E_{ER}}$	0.0	-0.06	-0.38	-0.06	-4.55
$\mathbf{E_{BA}}$	-0.53	0.0	-0.12	-0.72	-0.92
$\mathbf{E_{LANL}}$	-0.48	-0.10	0.0	-2.97	-3.90

Each value is the average cost of partitions produced by the method evolved for that application, minus the average cost of partitions produced by the partitioner listed at the top of the column. A negative value indicates that the evolved solution produces a lower average partition cost, with shaded cells indicating the difference is statistically significant at the $\alpha = 0.05$ level.

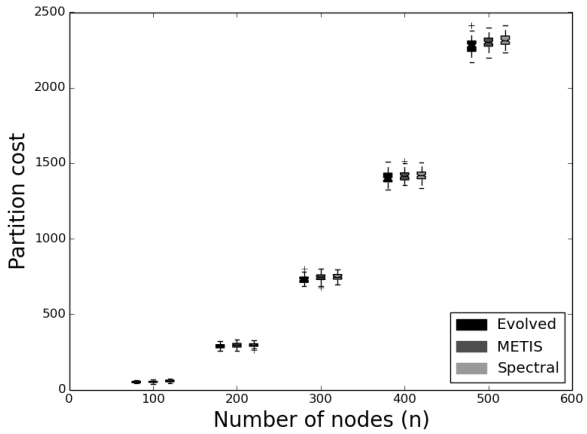
evolved solutions are specialized for their target graph type, they are also used to partition the other graph types and their relative performance is compared. The cost of the partitions produced by each method are compared pairwise using Wilcoxon rank-sum tests at a 95% significance. Finally, evolved solutions are used to partition graphs of various sizes to demonstrate their scalability compared to the general purpose partition solutions.

4.7 Results

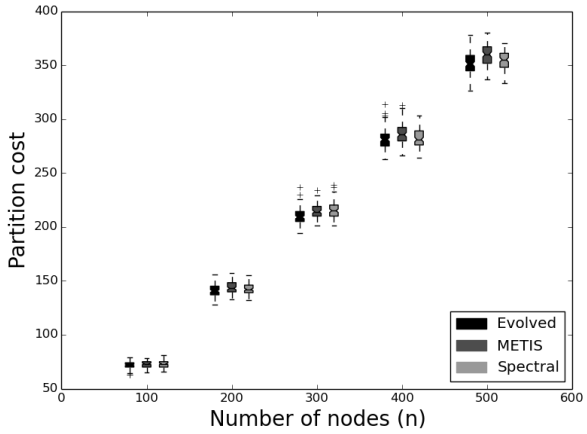
The parse tree representation of two sample evolved solutions can be seen in Figure 4.4, one for each of the random graph model applications. The evolved solutions for the LANL network application tend to be far more complex, and as a result, are too large to be included.

See Table 4.2 for the relative performance comparison of the evolved partitioning methods as well as METIS and spectral partitioning (SP). $\mathbf{E_{ER}}$, $\mathbf{E_{BA}}$, and $\mathbf{E_{LANL}}$, refer to the solutions evolved to target the Erdös-Rényi, Barabási-Albert, and LANL network graph sets, respectively. Each row compares the average partition cost of the method evolved for that application against each of the other partition algorithms. A negative value indicates that the evolved solution produces a lower average partition cost than the method indicated for that column. A shaded cell indicates the difference is statistically significant at the $\alpha = 0.05$ level.

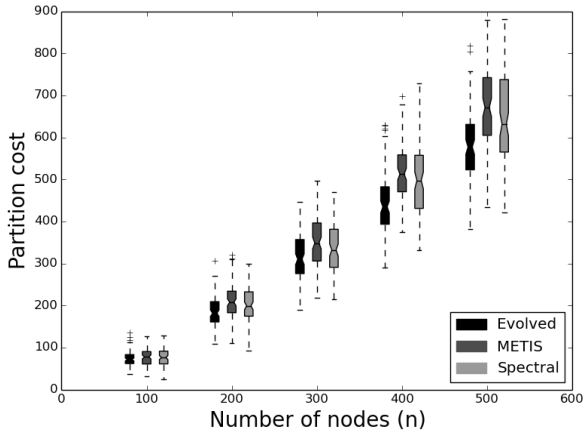
It is encouraging to see that for each graph type, the partitioner evolved for that type produces the lowest average cost, even if these differences are not always statistically significant.



(a) Erdös-Rényi



(b) Barabási-Albert



(c) LANL network

Figure 4.5: Cost of partitioning graphs of various sizes for each graph type.

The evolved methods consistently outperform the traditional spectral partition method. Compared to the off-the-shelf METIS software, the evolved solutions for the Barabási-Albert and LANL network graphs are also statistically superior. A notable exception is the evolved partition algorithm for the Erdös-Rényi application. The inability to statistically outperform METIS might be a result of the high randomness inherent to the Erdös-Rényi model, which might not consistently produce any graph characteristics that can be exploited during evolution.

The comparisons between the evolved solutions do suggest some amount of specialization has taken place, but the performance difference is not always statistically significant. $\mathbf{E_{BA}}$ and $\mathbf{E_{LANL}}$ do significantly outperform the $\mathbf{E_{ER}}$ on their targeted graph types. However, $\mathbf{E_{BA}}$ and $\mathbf{E_{LANL}}$ perform very similarly when interchanged. This could indicate that the graphs created from the LANL network data resemble graphs generated by the Barabási-Albert model; both evolved solutions might be taking advantage of characteristics common in both graphs.

Figure 4.5 shows the relative cost of partitioning graphs using the evolved partition solutions as well as METIS and spectral partitioning as the size of the graph grows. For each plot, the "Evolved" label refers to the partitioner that was evolved specifically for that graph type. Despite the fact that the solutions are evolved to target graphs with 100 vertices, the evolved partition algorithms still consistently outperform METIS and spectral partitioning as the size of the graphs increase.

4.8 Conclusion

Graph partitioning is a fundamental computer science problem with applications in many domain areas. Multi-level partitioning is a widely used state-of-the-art approach to efficiently approximate optimal partitioning. Although there are a variety of multi-level partitioning algorithms available, most are intended to serve as general purpose solutions. Some work has been done attempting to exploit common graph characteristics through the manual development of tailored solutions, but this tedious process must be repeated for each application. Even if a good set of specialized partition algorithms were available, it might not contain an adequate solution to an entirely new graph application.

This work addresses this limitation by employing genetic programming to automatically generate novel multi-level graph partitioning algorithms tailored to each application. The potential of this approach is demonstrated by evolving a set of algorithms, each tailored to perform well on graphs from a different source: two traditional random graph models and real world computer network subsets. These specialized solutions outperform traditional partitioning methods on their target graph types, and continue to do well as the size of the graphs increases. The platform implemented in this work can be quickly reapplied to any new application domains as they arise instead of relying on general purpose, off-the-shelf solutions.

Chapter 5

Automated Design of Network Security Metrics

Many abstract security measurements are based on characteristics of a graph that represents the network. These are typically simple and quick to compute but are often of little practical use in making real-world predictions. Practical network security is often measured using simulation or real-world exercises. These approaches better represent realistic outcomes but can be costly and time-consuming. This work aims to combine the strengths of these two approaches, developing efficient heuristics that accurately predict attack success. Hyper-heuristic machine learning techniques, trained on network attack simulation training data, are used to produce novel graph-based security metrics. These low-cost metrics serve as an approximation for simulation when measuring network security in real time. The approach is tested and verified using a simulation based on activity from an actual large enterprise network. The results demonstrate the potential of using hyper-heuristic techniques to rapidly evolve and react to emerging cybersecurity threats.

5.1 Introduction

In an age where new software vulnerabilities are discovered and exploited on a daily basis, best practices and fast response are insufficient to secure a large computer network. Administrators need to be able to understand, analyze, and track the level of security in networks they manage. As enterprise computer networks continue to grow in size and complexity, manual methods of analyzing network security are increasingly infeasible. Automated analysis tools are needed to highlight vulnerabilities and allow a pro-active defense strategy.

A common approach to analyzing computer networks is to model the network with a graph representation. Graphs can be used to model the physical or logical connectivity between computers on a network [1]. Alternatively, a graph might be used to represent the communication between networked machines [2]. Attack graphs are an example of a graph-based representation specific to security [5]. These graphs illustrate the potential paths an adversary can take to reach some compromise objective. Authentication graphs are a type of attack graph that can be used to identify the regions of a network an intruder can reach using stolen credentials [9]. Since graphs provide such a natural representation for networks, many network analysis techniques rely on graph-based heuristics.

This work demonstrates the feasibility of using hyper-heuristic techniques to automate the development of novel graph-based network security metrics. User activity data from a large real-world network is used to simulate network attacks that model adversaries traversing the network with stolen user credentials. These simulation results are used to guide the evolution of new graph heuristics that accurately predict attack success.

5.2 Network Authentication

Centralized single-sign-on systems, such as Kerberos [15], allow organizations to manage access control on a large scale. The credentials used to access a computer are often stored in a specialized cache on that machine. A variety of methods exist which allow an adversary to retrieve these credentials from a compromised computer [4]. Once the credentials have been obtained, they can be used to access and compromise other computers on the network. This entire process can be applied repeatedly, allowing an intruder to continue to traverse a growing portion of the network. The most notorious example of exploiting stolen credentials, known as *pass-the-hash*, abuses the weakness of reusable password hashes in older networks using Windows NT LAN Manager [16]. However, similar principles make this type of replay attack possible on modern systems as well, such as Kerberos [17].

5.2.1 Bipartite Authentication Graphs

The computers on a network and the user accounts that access them can be naturally represented as two independent sets of nodes in a bipartite authentication graph (BAG) [23, 22]. An edge in this graph connects a user node to a computer node and represents an occurrence where the user's authentication credentials are used to access the computer. This access can be direct (e.g., a user logging into a workstation) or indirect (e.g., through SSH or a remote desktop session). If the same account credentials are used to authenticate on additional computers, as is common in environments using centralized single-sign-on systems, then the corresponding user node will be adjacent to multiple computer nodes.

This graph representation makes it possible to identify the portions of a network which are vulnerable to credential theft attacks [9]. For example, if the computer $C1$ in Figure 5.1 is compromised, the credentials for user $U1$ could be stolen. The existing edges of the BAG indicate that the credentials for user $U1$ can also be used to access computers $C2$, $C3$, and $C4$. As a result, an adversary armed with the stolen credentials for user $U1$ would also be able to gain access to these additional computers.

Under normal circumstances, a computer's cache would only contain a subset of the credentials used to access the machine due to limits on the cache size or credentials being periodically removed. If the edges of a BAG incident to a given computer represent the authentication credentials assumed to be currently stored in that computer's cache, then upon compromise, the adversary can gain access to the credentials of all adjacent user nodes in the BAG. These new credentials could then potentially be used to access additional computers on the network. By repeating this process, the adversary can continue traversing the connected nodes, compromising a growing portion of the network.

5.2.2 Graph Heuristics

Because graphs are a natural way of representing computer networks, there are many examples of network applications that rely on graph heuristics. Minimal spanning tree heuristics are used to control network routing to avoid problematic cycles [3]. Graph partitioning methods are

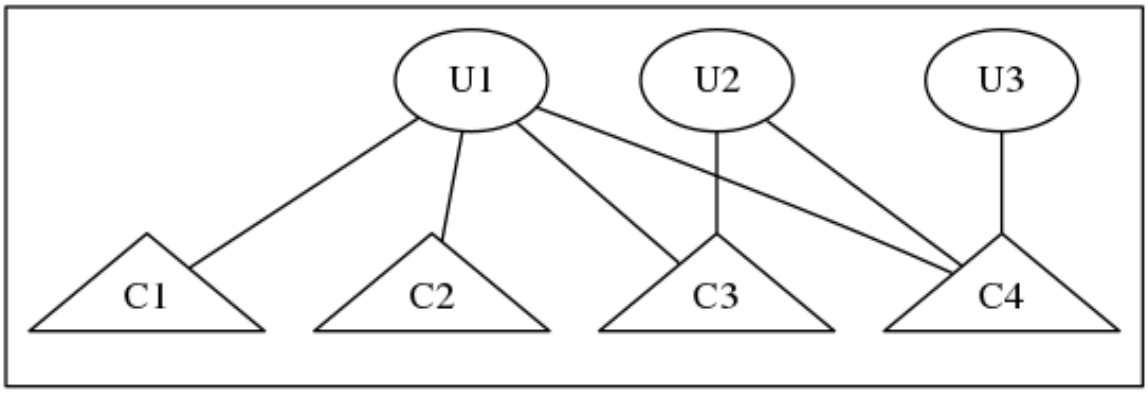


Figure 5.1: Example BAG with users U1, U2 and U3 and computers C1, C2, C3 and C4. An edge represents an authentication event between a user and a computer.

used to segment large computer networks to make it difficult for adversaries to penetrate the network [4, 9]. Path analysis heuristics are used on attack graphs to identify the likely routes attackers will use to compromise network resources [5, 28, 25, 29].

It is possible to achieve improved algorithm performance by using heuristics that exploit graph characteristics that are common in an application area [11]. Machine learning techniques have been used to automate the process of selecting the best heuristic for a problem from a set of available heuristics with high accuracy [6]. Unfortunately, this approach is limited by the quality and variety of the set of predefined heuristics; an optimal solution to a given problem cannot be selected if it is not already present in the heuristic set. Instead, domain expertise can be exploited to design novel customized heuristics tailored to a specific application. The process of designing new heuristics can be accomplished manually, but this can be difficult and time-consuming, often leading to an incomplete set of optimal heuristics. An alternative approach is to use hyper-heuristic machine learning techniques to automate the design and optimization of novel algorithms [7, 74].

5.3 Genetic Programming

Hyper-heuristics most commonly employ genetic programming (GP) to search a problem-specific space of algorithmic primitives. In GP, the solutions being evolved typically take the form of programs or heuristics. GP has been shown capable of automatically generating and

optimizing heuristics for problems in a variety of domains, including graph algorithm applications [58, 59, 10, 11]. A set of primitive operations is usually constructed by observing the common and essential elements of algorithms which have been designed to solve the intended problem. This primitive operation set is used as algorithmic building blocks by the GP to piece together new candidate algorithm solutions.

5.4 Related Work

There are many related works that employ graphs as an abstract representation of networks. Network connections, both physical and logical, can be modeled using graphs to visualize the network's topology [1, 3]. Communication between networked machines is also commonly modeled using weighted graphs where the edge weight represents the amount or frequency of communication between machines [2]. In particular, NetFlow communications lend themselves well to graph representations [75]; these provide a high-level, session-based view of the interaction between networked computers. Dynamic graph models have also been used to represent the changes or activity on a network over a period of time [76].

Graphs are particularly useful in network security applications. Graph partitioning methods have been used to segment networks to mitigate the damage potential of an intruder traversing the network [9]. Most graph partitioning techniques minimize the number or weight of edge removals needed to disconnect components of the graph; this feature can be leveraged to minimize the effort needed to segment the corresponding network or limit the impact on network user productivity. Attack graphs are another common abstraction method, this time for representing the avenues an adversary can take during a multi-step intrusion process [5]. Because of their usefulness for risk analysis and network hardening, automated methods of generating and evaluating these attack graphs have been developed [25].

This work builds on previous research that introduced the use of bipartite authentication graphs (BAGs) to model network user activity [22]. BAGs can be used in lieu of traditional attack graphs when the attack model is focused on traversal using stolen user credentials. An advantage of using BAGs over other attack graphs is that they can be constructed without

detailed information about the vulnerabilities on individual networked host machines. Authentication graphs can be constructed using logs from centralized authentication systems, which are often already being collected in enterprise networks. BAG representations have been used in previous work to identify anomalous user activity [77] and segment networks by finding minimal access control policy changes [9].

Although there are numerous examples of graph algorithms being applied to network security problems, many of these utilize general-purpose graph heuristics that do not exploit the specific characteristics of graphs that represent computer networks. Hyper-heuristics have been used to tailor heuristics to specific application domains [7], including those involving graph algorithms. Previous work investigated the use of hyper-heuristics to generate and optimize random graph generation heuristics that produce graphs with desirable characteristics, such as specific centrality distributions or community structures [10, 59, 58]. Customized graph partitioning heuristics have also been generated that improve upon the performance of general-purpose algorithms for targetted classes of graphs, including those representing computer networks [11].

5.5 Methodology

Network authentication events, consisting of a time stamp, a user account, and a network hostname, are used to construct a dynamic BAG. This graph provides the environment for a randomized credential theft and network traversal attack simulation. The simulation has two configurable settings meant to replicate authentication policy controls. The first parameter controls the duration a credential would be stored in a host's cache after an authentication event occurred. Repeated authentication events refresh this duration on existing credentials. If a credential is not refreshed before the configurable time limit is exceeded, the credential is rendered inactive, removing it from the host's cache. The second parameter controls the maximum size of the credential cache stored on the hosts. If an authentication event would add a credential to a cache that exceeds this limit, the oldest stored credential is removed to make room.

5.5.1 Lateral Movement Simulation

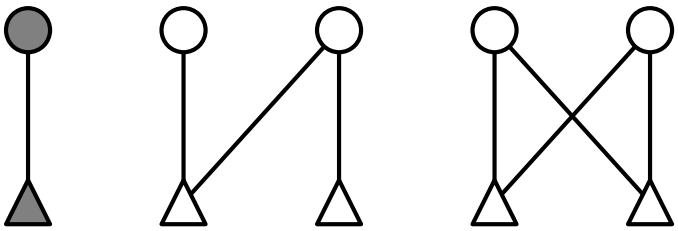
The attack simulation represents an adversary attempting to traverse the network with compromised user credentials. An adversary is initialized with a single compromised host. In this work, the initial host is chosen at random to represent a network computer inadvertently installing malware, potentially as a result of a phishing campaign. It is assumed that once a host is compromised, the adversary gains access to the credentials active on that computer.

At each subsequent time-step, the adversary will use any credentials they have accumulated to access additional hosts on the network. The simulation assumes the credential must be active on the additional hosts as a result of a legitimate authentication event for the adversary to compromise those computers. This behavior resembles a passive adversary attempting to hide their movement amongst legitimate activity in an effort to avoid detection. If an adversary chose to attempt access to a host not typically used by the impersonated user, the access is more likely to trigger a network intrusion alarm [78].

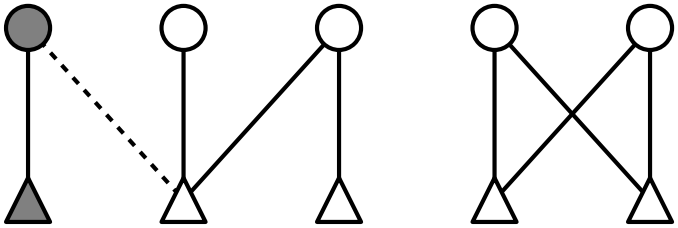
Whenever a new host is compromised by the adversary, any credentials on that host are added to the adversary's collection. This process is immediate, assuming the adversary is employing scripted exploit methods. Since the adversary is assumed to be automated, their traversal is not limited to a single host at a time. The adversary continuously harvests credentials from all compromised hosts simultaneously, seeking to compromise an ever-growing portion of the network. See Figure 5.2 for an example of lateral movement to additional network hosts. Once the configurable simulation time limit has expired, the set of compromised hosts is returned as an indication of attack success.

5.5.2 Compact Graph Representation

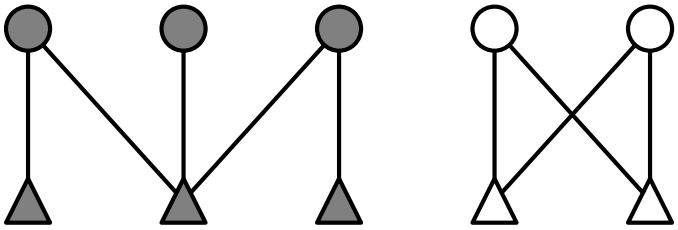
In order for the machine learning process to interact with reasonably sized graph representations, the dynamic bipartite authentication graph is used to produce a series of compact graphs that summarize the activity for 24 hour periods. This compressed representation assigns a weight to each authentication edge that is the count of the number of timesteps that authentication is active during a day. This representation is chosen because it presents the evolutionary



(a) Initial state



(b) New authentication edge



(c) After lateral movement

Figure 5.2: Lateral movement simulation example. Circles represent user credential nodes and triangles represent hosts. Compromised hosts and credentials are indicated by shaded nodes.

process with two interpretation options. The edge weight can be ignored, which would only consider the existence of authentication edges. Alternatively, the weights can be compared to differentiate between edges by their relative probability to be active.

5.5.3 Hyper-Heuristic Approach

A population of graph heuristics is evolved to predict the chance of simulated attack success given a compact authentication graph model.

Representation: Candidate solution algorithms are represented using strongly-typed genetic programming (GP) parse trees [61].

Initialization: An initial population of parse tree solutions is randomly constructed from the available input and operation nodes. A configurable maximum height parameter is used to limit the size of the initial parse tree solutions. Ramped half-and-half solution generation is used, which produces full parse trees of maximum height for half the population and variable height trees (up to the maximum) for the remainder.

Evaluation: Solutions are evaluated by comparing their output to the attack simulation results. For each day of activity, the attack simulation result is compared to the return value of the individual heuristic. The absolute percentage differences between these values are summed and normalized by the number of days to find the error rate of the prediction heuristic. This error rate is negated and used as the fitness for the solution, as shown in Equation 5.1. The evolutionary process attempts to maximize these fitness scores, producing solutions with low error rates. Solutions that fail to return a result within a configurable time limit have their fitness values set to negative infinity to discourage inefficient solutions.

$$fitness = -\frac{\sum_{d \in days} \left| \frac{simulated\_result - predicted\_result}{simulated\_result} \right|}{|days|} \tag{5.1}$$

Parent Selection: Parents are selected by taking a random sample of size k from the population and choosing the solution from the sample with the best fitness. This is known as k-tournament selection.

Recombination: Due to the destructive nature of parse tree variation operators, offspring are generated using either recombination or mutation, not both [57]. If a pair of parent solutions are selected for recombination, two offspring are produced using random subtree crossover.

Mutation: If recombination is not selected, an offspring is created by cloning a single parent, then performing random subtree replacement.

Survival Selection: Elitist truncation is used for survival selection, simply selecting the solutions with the best fitness.

Termination: Execution of the GP is terminated when a configurable number of generations have passed without any improvement in the average population fitness (convergence threshold).

Parameters: The parameters for the GP can be seen in Table 5.1. Aside from the convergence threshold, these values were programmatically tuned by a random-restart hill-climbing search attempting to optimize the best fitness found during evolution. This tuning process was also used to choose the parent and survival selection techniques. Other selection methods considered include fitness proportional and uniform random selection. The possible values for these parameters were inspired by previous work evolving graph algorithms [11, 10]. The convergence threshold was hand tuned to ensure termination in the time available.

Table 5.1: GP Parameter Values

Parameter	Value
Population size	400
Offspring per generation	600
Parent selection tournament size	8
Minimum initial parse tree height	4
Maximum initial parse tree height	7
Recombination probability	70%
Mutation probability	30%
Convergence threshold	10

5.5.4 Primitive Operations

The following categories of operations make up the set of primitives available to the GP. These operations were inspired by previous work on authentication graph analysis [23, 9] and the automated design of graph-based heuristics [11, 10, 62, 58].

Math operators: Basic addition, subtraction, multiplication, division, modulus, exponentiation, additive and multiplicative inverse. Some of these operators require special attention due to the stochastic nature of the process. For example, if division would produce a division by zero exception, it instead divides by a value very close to zero. These include operations that reduce a set of values to a single value, such as *sum* and *average*.

Numerical constants: Return a constant integer ($\{0, 1, 2, \ldots, 10\}$) or probability ($\{0.001, 0.01, 0.1, 0.2, 0.3, \ldots, 1.0\}$) value randomly chosen once during initialization. These possible values were inspired by previous work evolving graph heuristics [11, 10].

Boolean nodes: True and false constant nodes, as well as a node that randomly returns true according to an input probability.

Control flow: Standard *if-then-else* style conditional branching, as well as *for* and *while* loops.

Global graph metrics: Metrics based on the entire graph, such as average degree, number of nodes, or graph diameter.

Graph elements: Collections of graph edges and nodes.

Graph element metrics: Metrics associated with graph elements, such as node centrality values or edge weights.

Maps: Map a collection of elements to their respective metrics. For example, the betweenness centrality of a set of nodes.

Collection manipulation: Manipulate collections, such as concatenation or conditional filtering.

Subgraph induction: Induces a subgraph from a collection of nodes or edges.

Table 5.2: LANL Authentication Dataset Details

Unique Users	10,044
Unique Computers	15,779
Unique (User, Computer) Pairs	124,020
Total Authentication Events	101,918,344
Average Daily Authentication Events	2,547,958.6

5.6 Experiment

Dynamic BAGs are constructed from the authentication data produced by Los Alamos National Laboratory (LANL) [44]. One such BAG is produced for every day for the first forty days of activity in the dataset. A summary of this data can be seen in Table 5.2. Figure 5.3 shows the daily number of authentication events with an obvious weekly pattern. Presumably, the valleys correspond to weekends, but it is interesting to note that most of the weeks only show a single day with dramatically fewer authentication events. This could be the result of automated processes, such as patch management services, running on a weekly basis during employee downtime to minimize network user impact. Unfortunately, the anonymized dataset does not contain enough information to better explain this pattern.

The lateral movement attack model is simulated for each day for two possible credential policy configurations. The first assumes a maximum cache size of ten, removing the oldest cached entries to make room for new credentials. The second configuration removes credentials from host caches after one hour. Simulations are initialized by populating the graph with twelve hours of network activity. After initialization, the simulated attack begins with a single, randomly chosen point of compromise. The simulation proceeds for one hour and the degree of success is measured by the percentage of the networked hosts that are compromised upon termination. See Figure 5.4 for an illustration of how the the percentage of the network compromised tends to grow over the course of the attack simulation. The upward trend shows how the portion of the network compromised grows during the simulation. The high variation and skew are a result of the inclusion of days with very low authentication activity; this dramatically limits an adversary's ability to reach a large portion of the network.

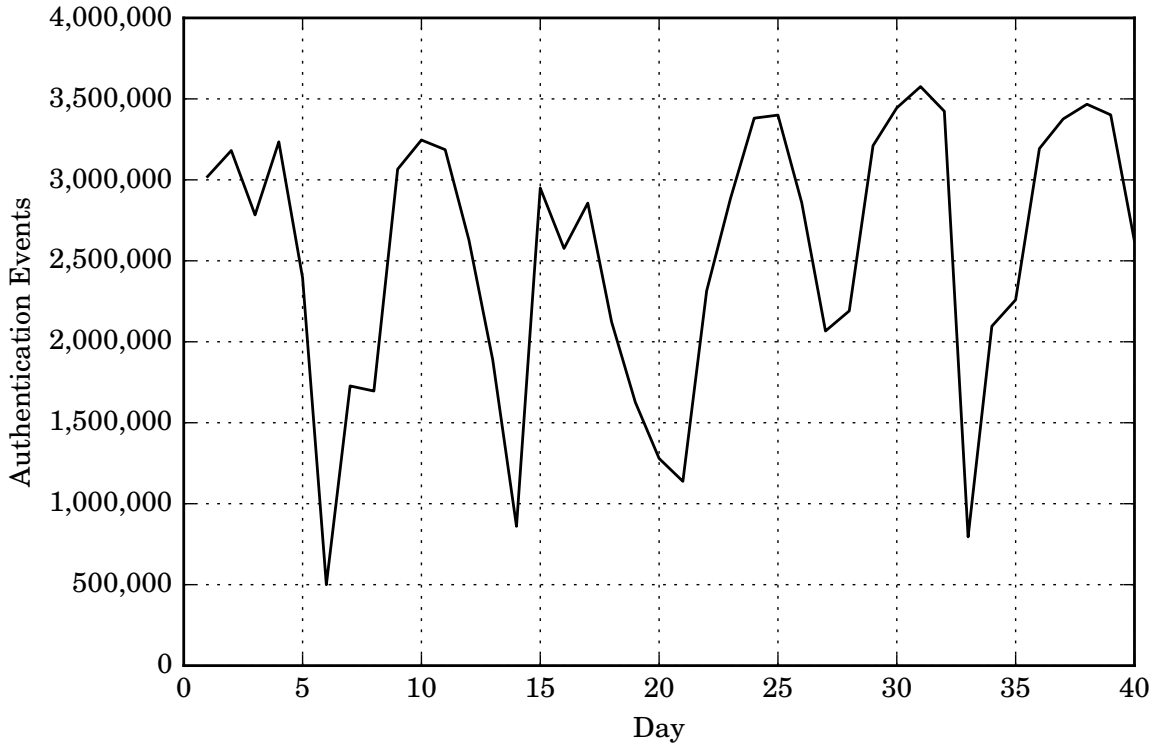


Figure 5.3: Count of authentication events per day.

This success measure for each day is averaged over one hundred repeated simulations, each with a different initial point of compromise. Figure 5.5 shows the final average simulation outcome for each day and each policy configuration. Again, the weekly pattern is obviously present. Compared to the ten credential limit policy, the one hour credential expiration policy consistently reduces the success of the adversary's compromise percentage. Further examination of the graphs produced by this data suggests that this is due to authentication edges that connect computers accessed infrequently by a small number of users; these edges tend to remain active for long periods of time within the simulation.

The first thirty days of the dataset are used to train the GP. A compact graph representation of each day's activity is created as described in Section 5.5.2. Figure 5.6 shows the distribution of authentication edge activity levels for both policy configurations. Edges near the lower end of the horizontal axis are only active for short periods during the simulation. Alternatively, edges near the upper end are active for the majority of the simulated period. The dramatic difference between these distributions, especially at high edge activity levels, illustrates the impact of the

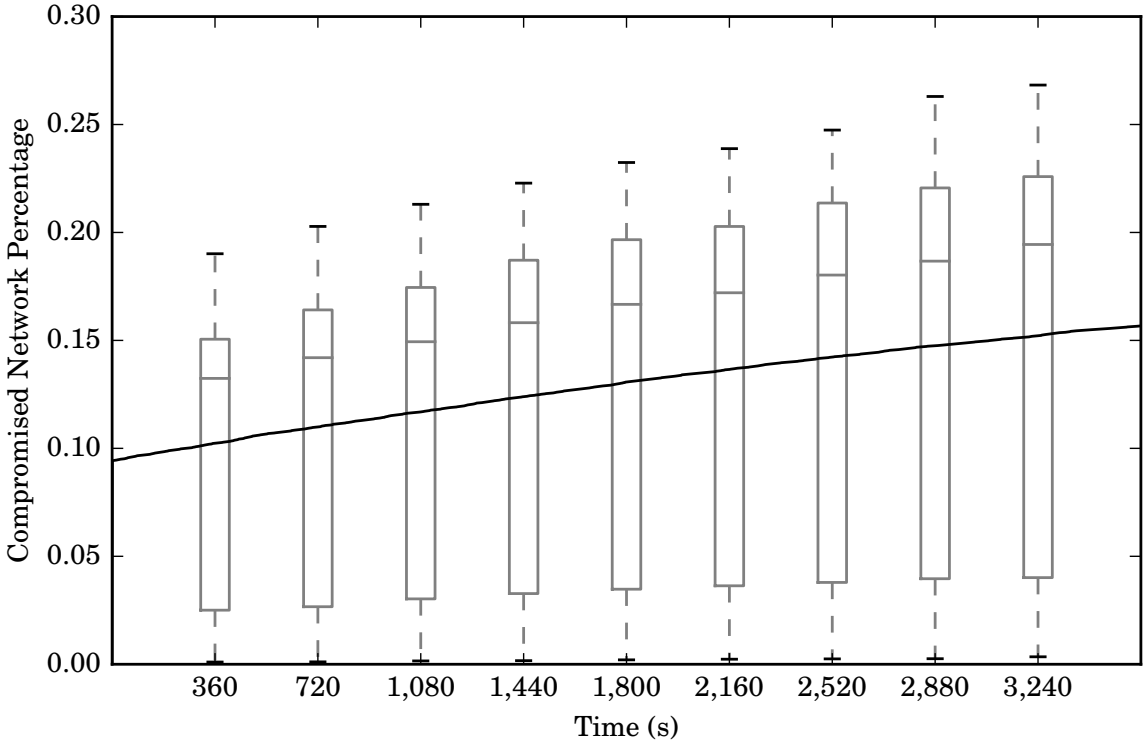


Figure 5.4: Mean simulation results using 1-hour ticket lifetime policy. The vertical axis shows the percentage of the network compromised, averaged over all adversaries, at each time step. The boxes indicate the variation between different days.

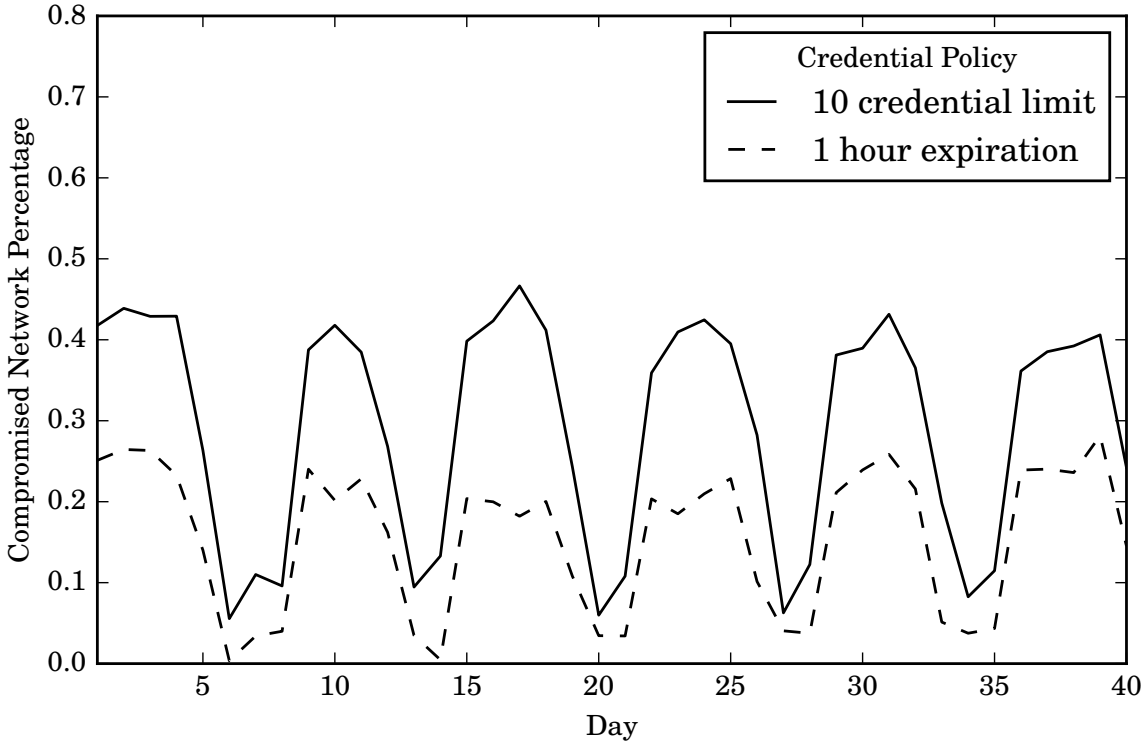


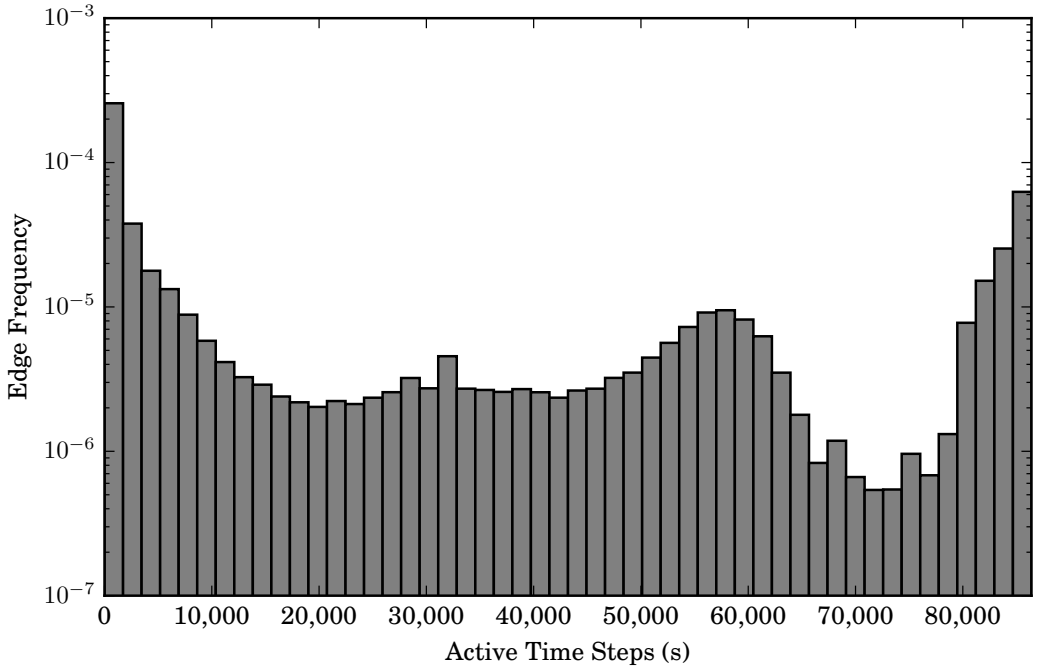
Figure 5.5: Daily simulation results for both credential policies.

policy configuration selected. These edge activity levels are available to the evolved heuristics as edge weights, allowing some of the temporal information to be leveraged despite the static nature of the compact graph representation. For each policy configuration, a population of heuristics is evolved to predict the simulation outcomes. Additionally, a third population is trained to predict the simulation outcomes for both policy configuations simultaneously. This is done to examine the benefit of focusing on specific credential policies instead of seeking a more generalized heuristic. The performance of the best final evolved solutions from each population is measured against the simulation results for the final ten days for validation.

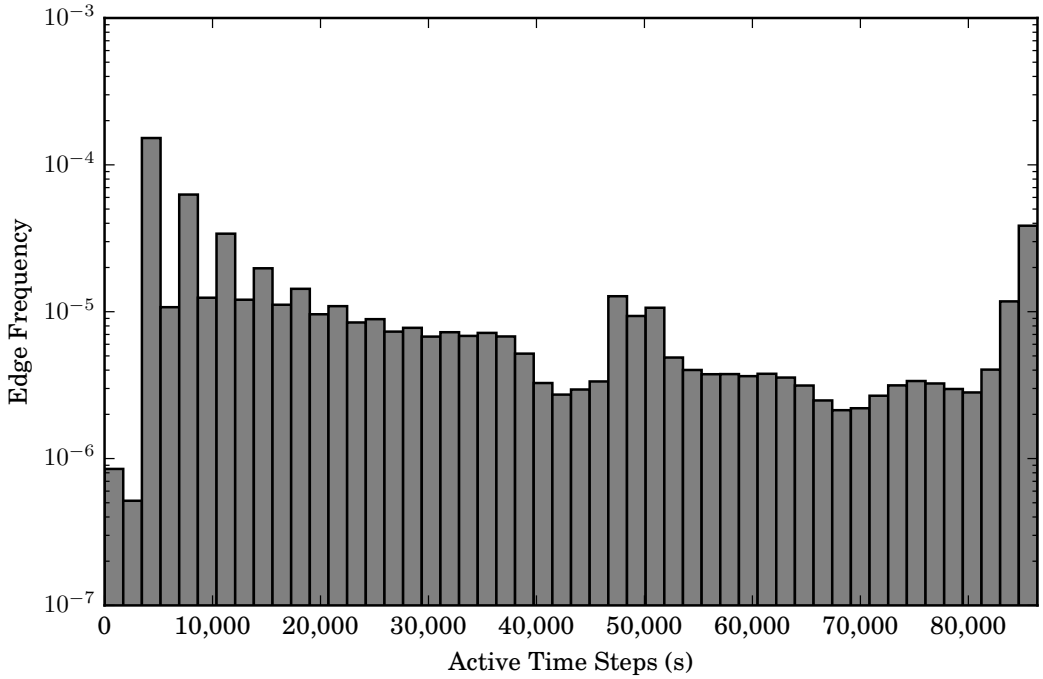
5.7 Results and Discussion

Although the evolved heuristics are too complex to be included here (the smallest being over 200 lines of code), some functional elements that commonly occur are:

1. Induce a subgraph with the most active (highest weight) edges



(a) 10 credential limit policy



(b) 1 hour expiration policy

Figure 5.6: Distribution of authentication edge activity levels for compressed graphs for both credential policies. Low values (to the left) indicate edges that are rarely active in the original dynamic graph. High values indicate edges that are active the majority of that day. Note that the vertical axes are log scaled.

Table 5.3: Comparison of Evolved Metric Heuristics

	10 Credential Limit		**1 Hour Expiration**	
Method	**Result**	**Error**	**Result**	**Error**
Simulation	29.797%	N/A	17.484%	N/A
GP-A	29.151%	6.15%	21.411%	60.93%
GP-B	27.093%	14.85%	17.571%	11.23%
GP-C	26.427%	28.00%	20.387%	71.79%

2. Find the connected components in the induced graph

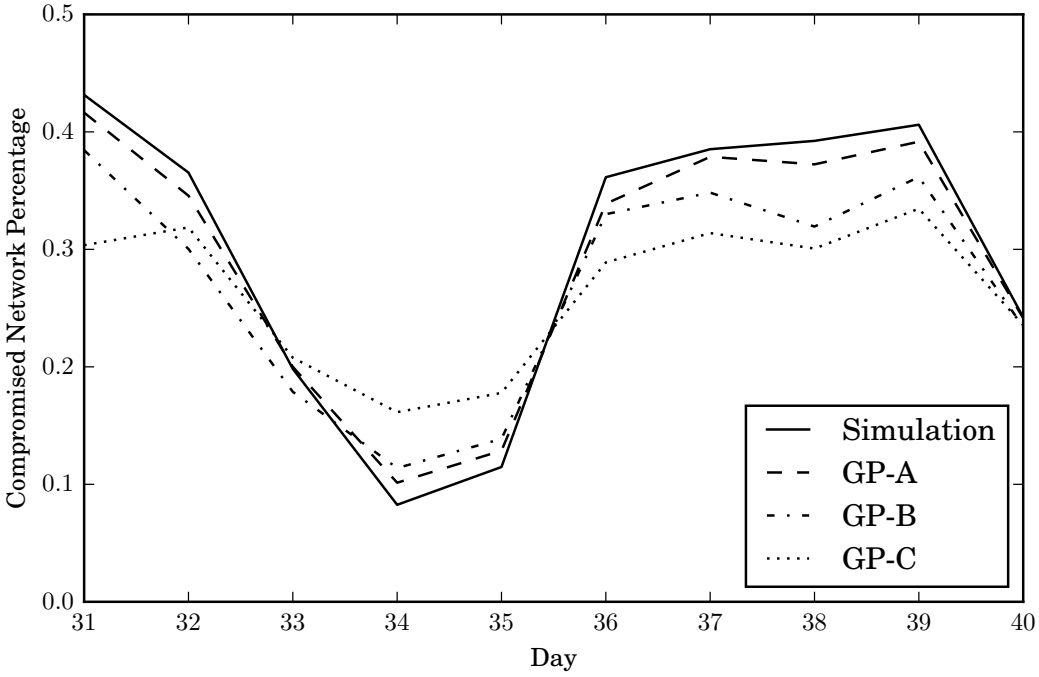
3. Filter out the account vertices in each component vertex set

4. Return a value based on the number of computers in each component relative to the number of computers in the original graph

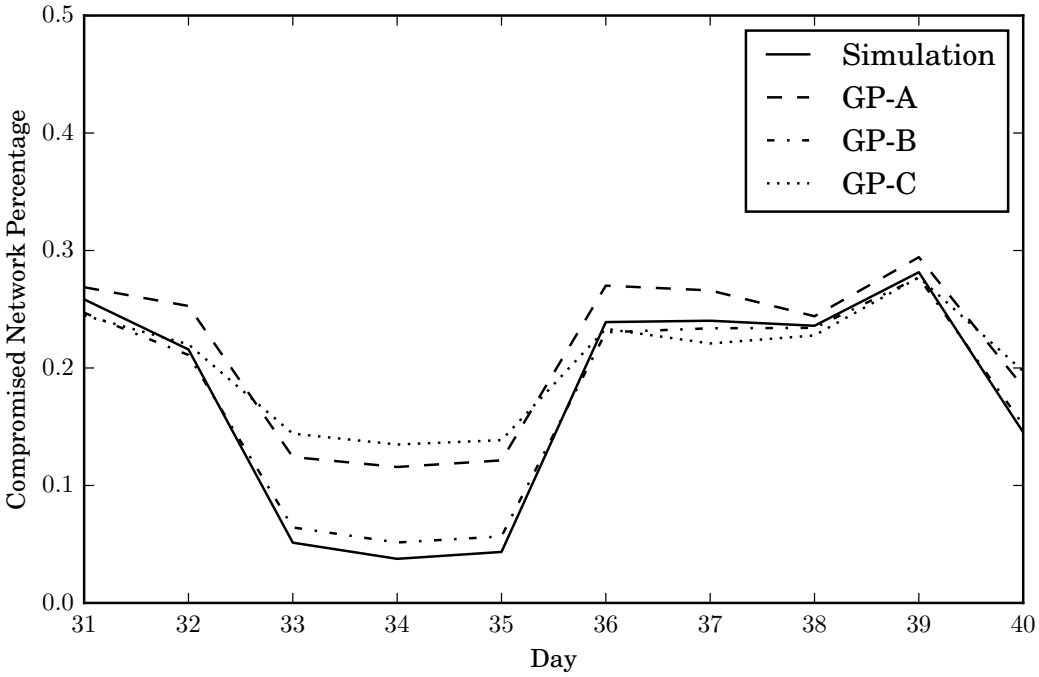
This is not surprising, considering how the connected components represent portions of the network which can be traversed with lateral movement.

Table 5.3 shows the simulated and predicted compromise percentages averaged over each day of the validation data. **GP-A** refers to the best heuristic trained using the ten credential limit policy. Similarly, **GP-B** indicates the heuristic produced by the one hour expiration policy. **GP-C** is the heuristic evolved to predict the combined simulation results. Figure 5.7 shows the daily comparison of the simulation results and predicted values for both credential policies. The distance between the predictions of the evolved heuristics and the simulation results is an indicator of the quality of the heuristic. Superior solutions lie closer to the simulated outcomes. In both policy configuration cases, the heuristic evolved to target that configuration (**GP-A** for Figure 5.7a and **GP-B** for Figure 5.7b) more accurately tracks the simulation results. The results suggest that it is beneficial to target the particular credential policy in use instead of seeking a more general purpose solution.

One interesting outcome is that **GP-C**, which represents a "hybrid" approach, tends to perform worse than both of the more targeted heuristics; this is especially evident in Figure 5.7a. It is understandable for this attempt at a generalized heuristic to be performing worse than the appropriately targetted heuristic, but it is surprising that, in many cases, it is also outperformed by the heuristic targetting the wrong credential policy. One possible explanation is that the



(a) 10 credential limit policy



(b) 1 hour expiration policy

Figure 5.7: Comparison of simulated results and predictions from each evolved solution for both credential policies. Higher quality solutions lie closer to the simulation results.

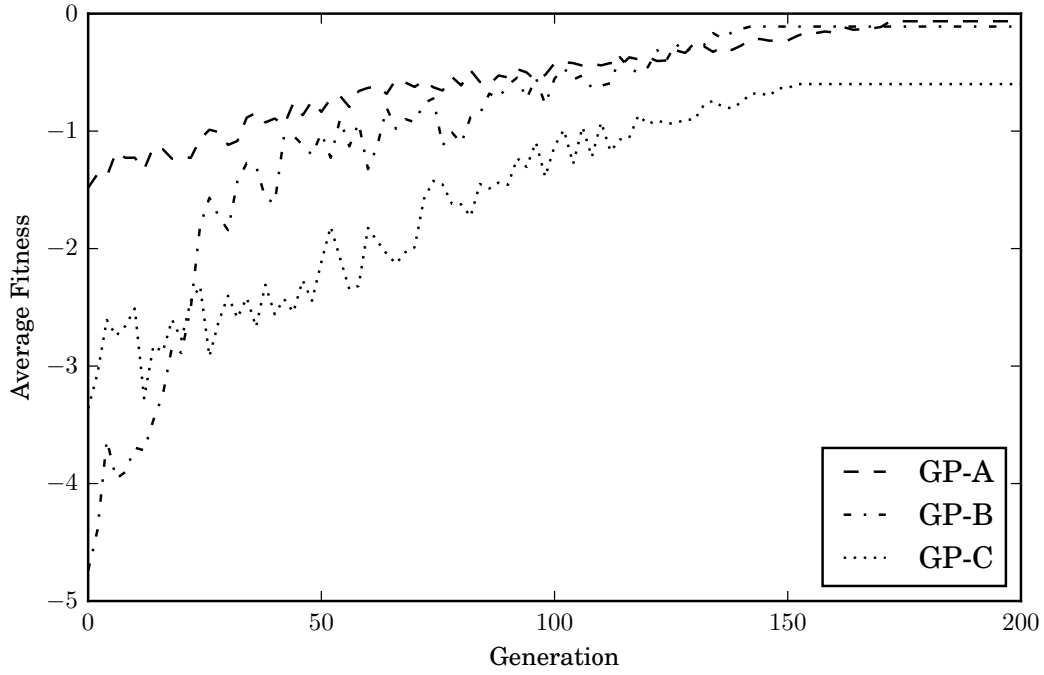


Figure 5.8: Population fitness value vs. generation for each evolved population. Fitness is averaged over twenty repeated executions of the GP.

differences between the compact graph representations for the two policy configurations made fine-tuned exploitation of specific graph properties difficult. Regardless of the true reason for the degraded performance, the results demonstrate the value of evolving heuristics tailored to the specific policy.

Figure 5.8 shows the mean population fitness for each generation, averaged over twenty repeated executions of the GP. Progress is indicated by an upward trend. The low initial fitness for the population of **GP-A** is likely the result the higher variation in the simulation results for the 10 credential limit policy. Evolution quickly overcomes this disadvantage, however, and both targeted GPs converge on similar fitness values. Aside from very early generations, **GP-C** consistently has lower population fitness compared to the other GPs.

The results presented here demonstrate the potential of hyper-heuristics to automate the development of novel network security metrics. In this work, evolution was guided by an attack simulation, but this could be replaced with data from penetration testing or genuine compromise events. Institutions with large computer networks are likely already collecting the data needed to train these heuristics. The approach presented in this work could provide system

administrators a new capability to better leverage this data without relying on expert knowledge of the specifics of an adversary's techniques. This has the potential to reduce the time needed to understand and react to emerging threats.

5.8 Conclusion

In the ever-evolving world of cybersecurity, system administrators need new ways to understand and visualize risks and vulnerabilities. Manual analysis can be prohibitively expensive and time-consuming, limiting our ability to react to new adversary techniques. This work has demonstrated the potential of hyper-heuristic techniques for the automated development of network security metrics. Evolved heuristics were able to accurately predict simulated attacks on network models based on real-world data for a complex network. Automated design can improve our security capabilities enabling us to rapidly react to emergent threats with less reliance on subject matter experts. Although the current results are focused on computer networks, the approach could be easily extended to include physical domain elements for more comprehensive security.

Chapter 6

Automated Design of Random Dynamic Graph Models

Dynamic graphs are an essential tool for representing a wide variety of concepts that change over time. Examples include modeling the evolution of relationships and communities in a social network or tracking the activity of users within an enterprise computer network. In the case of static graph representations, random graph models are often useful for analyzing and predicting the characteristics of a given network. Even though random dynamic graph models are a trending research topic, the field is still relatively unexplored. The selection of available models is limited and manually developing a model for a new application can be difficult and time-consuming. This work leverages hyper-heuristic techniques to automate the design of novel random dynamic graph models. A genetic programming approach is used to evolve custom heuristics that emulate the behavior of a variety of target models with high accuracy. Results are presented that illustrate the potential for the automated design of custom random dynamic graph models.

6.1 Introduction

Graphs are a powerful and flexible method of representing a wide variety of concepts where the relationships between objects are a critical element [51]. Common applications include utility and transportation grids as well as computer and social networks. Because graphs are such a versatile way to represent and store data, countless algorithms exist that operate directly on graph structures to solve problems. Graph partitioning algorithms are used to efficiently distribute parallel computation jobs [65]. Social networks use graph-based community detection approaches to improve automated recommendations [79].

When developing new graph algorithms, researchers often turn to random graph models to test and demonstrate the flexibility and scalability of their solutions. Random graph models are also an invaluable tool for anticipating the development of a network, such as predicting the spread of a communicable disease [80]. Regardless of the specific application, the proper selection of a random graph model is critical. An inappropriate model will produce graphs that can differ dramatically from the intended target and provide an unrealistic representation. For example, a model that produces graphs that resemble transmission grids will probably be unsuitable for representing social networks.

A variety of graph similarity metrics exist that can be used to select the most appropriate model [59]. However, this approach only works if a good set of models is available a priori. A new model can be developed to suit a specific application, but manual development can be difficult and time-consuming [46]. Hyper-heuristic search techniques [7] have been used to automate the design of generative random graph models [10, 58].

Random dynamic graph models are a trending research topic, but the field is still relatively new [81, 82]. Previous methods of automating the design of random graph models are limited to static graphs by design [10, 58, 83]. This work investigates extending the use of hyper-heuristics for automated graph model generation to dynamic graph applications. Genetic programing (GP) [57] is used to evolve heuristics that capture the behavior of a variety of random dynamic graph models. Results show that evolution is able to capture the characteristics of the target models with a high degree of accuracy.

6.2 Background

This section reviews a fundamental random graph model as well as a variation of this model capable of producing dynamic graphs. The applications targeted in this work build upon this extended model. Also covered is some background on automated algorithm design using a hyper-heuristic search.

6.2.1 Erdös-Rényi Model

The Erdös-Rényi graph model is one of the most basic random graph models, but it is also the most studied [47, 48]. This simple model takes two parameters: the number of vertices n and a probability p. Any possible edge between two vertices in the graph will exist with an independent probability p. The model is typically denoted as $G(n, p)$.

6.2.2 Dynamic Erdös-Rényi Model

Previous work introduced an extension of the Erdös-Rényi graph model that can be used to create dynamic graphs [81]. This extension adds two additional model parameters: α and β. The initial graph is created according to the static Erdös-Rényi model. At each time step, missing edges are added with probability α and existing edges are removed with probability β. If α and β are constant, the number of edges in the graph will tend towards $\binom{n}{2} * \frac{\alpha}{\alpha+\beta}$. See Zhang et. al. [81] for more detail on the characteristics of this model.

6.2.3 Hyper-Heuristics

Instead of attempting to solve a specific problem instance, a hyper-heuristic search aims to find a heuristic solution that can produce high-quality solutions to a class of problems [7]. This work leverages genetic programming (GP), a common hyper-heuristic technique, to evolve a population of programs that modify an input graph in a way that resembles a target random dynamic graph model. Solutions are represented using traditional Koza-style parse trees [57] with strongly-typed versions of tree construction and variation [61].

6.3 Related Work

Automating the design of static random graph models is well studied. Bailey et. al. demonstrated that GP could be used to automate the inference of graph models for complex networks [58]. This approach was extended with increased primitive granularity to achieve more flexibility in random graph generation [10]. Harrison et. al. investigated the impact of objective selection when using GP to evolve random graph models [83]. The evolution of graph

models has also been extended to directed graph applications [84]. Menezes et. al. employed a symbolic regression GP approach to select edges to add to incrementally build a network [85]. Methods other than GP have been used to automate the design of graph models, such as using simulated annealing to optimize an action-based approach to construct complex network models [86].

A key limitation of these approaches is that they are designed to generate static graph models. Many of these methods start with an empty initial graph and aren't capable of modifying an existing graph. Since most of these methods are focused on iteratively building graphs, they don't incorporate functionality to remove existing edges. This work aims to extend automated graph model design to create graph update heuristics that accurately capture dynamic graph behavior.

6.4 Methodology

This section describes the approach used in this work to evolve heuristics for the generation of dynamic random graphs.

6.4.1 Representation

Solutions are represented using strongly typed parse trees [61]. These trees are constructed from primitive graph-based operations that are described in Section 6.4.4. Solutions in the initial parent pool are randomly generated from the available operations using a ramped half-and-half approach. See Figure 6.1 for an example parse tree representation of a basic random graph heuristic.

6.4.2 Evaluation

To evaluate the quality of an evolved solution, its behavior is compared to that of a target dynamic graph model. An initial graph is constructed according to the target model and duplicated for comparison. The target model is used to update the initial graph by adding and/or removing edges. Similarly, the evolved solution is used to update the duplicate graph. The number of edges added and removed from both graphs is tracked along with the total number of edges.

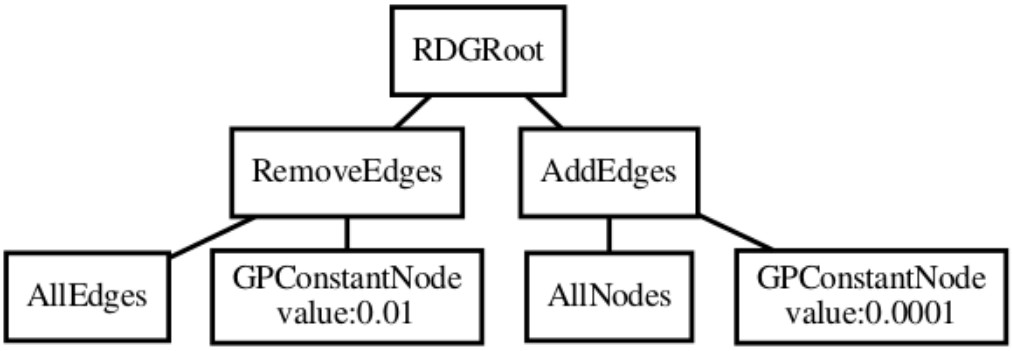


Figure 6.1: Example random graph heuristic parse tree that first removes 1% of existing edges, then adds new edges with probability 0.01%.

This process is repeated for a configurable number of time steps to produce a final graph for both the target model and the evolved solution.

The distribution of vertex degrees of the final output graphs are compared to measure the similarity between them. A two-sample Kolmogrov-Smirnov (KS) distribution comparison test is used to compare the sample distributions for both graphs. This test returns a p-value in the range $[0, 1]$ that is maximized when the samples are similar and likely to have come from the same distribution. **DC** (degree centrality) is used to refer to the p-values from this KS test comparison.

The distributions of the number of edges added and removed at each time step are compared in a similar fashion to calculate the terms **EA** (edges added) and **ER** (edges removed). The final fitness component measures how well the evolved heuristic mimics the target model with respect to the number of edges in the graph at each time step. This metric, refered to as **SD** (size difference), is defined as

$$\mathbf{SD} = max\left(1 - \frac{1}{T}\sum_{t=1}^{T}\frac{|size(G_t) - size(H_t)|}{size(G_t)}, 0\right) \tag{6.1}$$

where $size(G)$ is the number of edges in graph G, T is the configurable number of time steps per evaluation, G_t is the graph produced by the target model at time step t, and H_t is the graph produced by the evolved heuristic at time step t. The absolute difference between the size of the two graphs is normalized by the size of the target and averaged over all time steps. This

value is subtracted from one to convert it to a maximization objective. Values for this objective below zero are set to zero to keep each objective on the same $[0, 1]$ scale.

To make it easier to compare evolved objective scores across applications, each objective score is scaled using the objective value achieved by comparing a model against itself using the formula

$$\mathbf{\Theta} = 1 - \frac{|\mathbf{\Theta_T} - \mathbf{\Theta_E}|}{\mathbf{\Theta_T}} \tag{6.2}$$

where $\mathbf{\Theta}$ is an objective in $\{\mathbf{DC}, \mathbf{EA}, \mathbf{ER}, \mathbf{SD}\}$, $\mathbf{\Theta_T}$ is the value for that objective achieved by the target model evaluated against itself, and $\mathbf{\Theta_E}$ is the value for that objective achieved by the evolved model. This has the added benefit of penalizing overfit solutions that mimic the evaluation test cases better than the model fits itself.

The entire evaluation process is repeated for a configurable number of test cases to measure the robustness of the evolved graph model. Final solution fitness is defined as

$$fitness = \frac{1}{C} \sum_{i=1}^{C} \frac{\mathbf{DC} + \mathbf{EA} + \mathbf{ER} + \mathbf{SD}}{4} \tag{6.3}$$

where C is the configurable number of test cases per evaluation.

The four metrics previously described are used to produce a single fitness value. Alternatively, they could be used as separate objectives in a multi-objective approach. In this proof-of-concept, the single fitness value is used to assist with interpretability of the results and selecting exemplar solutions without problem-specific knowledge. Future work will leverage a multi-objective optimization approach.

During evaluation, fitness is calculated incrementally after each test case. If a solution's fitness is in the bottom quartile compared to the population after a configurable minimum number of evaluation test cases, the evaluation process is terminated early. This is done to avoid wasting expensive evaluation time on obviously inferior solutions.

6.4.3 Evolution

To better leverage parallel computation resources, this work employs an asynchronous evolutionary approach. In the initial phase, parent solutions are generated randomly until enough

of them have been evaluated to form a starting population. Subsequent solutions are added to the population one at a time as they complete evaluation. After adding a newly evaluated solution to the population, an inverted k-tournament (selecting the lowest fitness) removes a solution from the population. Then, a new offspring is generated either by sub-tree crossover with two parents or sub-tree mutation from a single parent. Parent solutions are chosen using traditional k-tournament selection. The new offspring is then added to the asynchronous queue for evaluation. This process continues until a configurable number of evaluations have been completed.

6.4.4 Primitive Operations

As this work employs a strongly typed GP approach, each instance of an operation has an associated type to enforce compatibility. The available primitive types are as follows:

Boolean: returns a boolean value (true or false)

Integer: returns a whole number

Float: returns a floating point number

Probability: returns a floating point number bound to the range $[0, 1]$

Numeric: pseudo-type that refers to operations that can handle Integer, Float, or Probability types (e.g., Add)

NodeList: a collection of nodes in the input graph

EdgeList: a collection of node pairs from the input graph

List: pseudo-type that refers to operations that can handle both NodeList or EdgeList types

GraphOp: an operation that, instead of being used for a return value, alters the input graph

NodeOp: an operation that takes a node input when called and alters the input graph

Op: pseudo-type that refers to operations that can handle GraphOp, NodeOp, or EdgeOp types

Root: a special primitive type only used for the root node

Note that all references to a pseudo-type (Numeric or List, or Op) must match for an instance of a primitive operation. For example, all the List types must match for an instance of the ListIntersection operation; this primitive cannot find the intersection of a NodeList and an EdgeList.

All evolved solutions begin with a special root node primitive. This primitive has one, two, or three GraphOp children that it calls sequentially. In addition to altering the input graph through the actions of its children, this primitive also tracks and returns the edges added and removed from the input graph during execution of the parse tree. See Table 6.1 for a description of the rest of the primitive operation set.

The primitive set used was initially inspired by previous work evolving static random graph models [10, 83, 58]. Some operations were added specifically to ensure the primitive set was capable of capturing the behavior of the application models targeted in this work. This primitive set is fairly large, mostly due to the strongly-typed genetic programming approach used. Future work will investigate how well the heuristic search makes use of each primitive in an attempt to prune unnecessary operations.

6.4.5 Parameters

Table 6.2 lists the values of the configurable parameters used in this work. These parameter values were initially inspired by previous work evolving random graph models, but they have been hand-tuned to improve performance for this application.

6.5 Experiment

The Dynamic Erdös-Rényi model described in Section 6.2 is used to create a variety of target application models. For each target model, a population of heuristics is evolved to mimic the model's behavior. The target models are created by manipulating the α parameter. All target models use the same values for the other model parameters: $n = 1000$, $p = 0.01$, $\beta = 0.03$.

Table 6.1: Primitive Operations

Primitive	Type	Inputs	Description
SequentialOp	Op	**O1**,**O2**[,**O3**]:Op	sequentially executes two or three subtrees
NoOp	Op	None	does nothing
ForNodeLoop	GraphOp	l:NodeList, **N**:NodeOp	for each node n in l, execute **N**(n)
ForIndexRange	GraphOp	i:Integer, **G**:GraphOp	execute **G** i times
CreatePath	GraphOp	l:NodeList, b:Boolean	connect subsequent nodes in l to create a path; if b, connect first and last nodes in l to create a cycle
ConnectToNodes	NodeOp	u:Node, l:NodeList, p:Probability	for every node v in l, connect u and v with chance p
AddEdges	GraphOp	l:NodeList, p:Probability	for every pairing of nodes in l, connect with chance p
RemoveEdges	GraphOp	l:EdgeList, p:Probability	for each edge in l, remove with chance p
RewireEdges	GraphOp	l:EdgeList, p:Probability	for each edge in l, rewire with chance p
AddPairwiseEdges	GraphOp	$l1, l2$:NodeList	connect node pairs at each index in lists $l1, l2$ with chance p
CreateTriangles	GraphOp	$l1, l2, l3$:NodeList, p:Probability	add edges to create triangle with nodes at each index in lists $l1, l2, l3$ with chance p
IfOp	GraphOp	b:Boolean, **G**:GraphOp	if b, execute **G**
IfElseOp	GraphOp	b:Boolean, **G,H**:GraphOp	if b, execute **G**, else execute **H**
Add	Numeric	x, y:Numeric	returns $x + y$
Subtract	Numeric	x, y:Numeric	returns $x - y$
Multiply	Numeric	x, y:Numeric	returns $x * y$
SafeDivide	Numeric	x, y:Numeric	returns 1 if $y = 0$, else x/y
Modulus	Numeric	x, y:Numeric	returns $x\%y$
Not	Boolean	x:Boolean	returns $\neg x$
And	Boolean	x, y:Boolean	returns $x \cap y$
Or	Boolean	x, y:Boolean	returns $x \cup y$
LessThan	Boolean	x, y:Numeric	returns $x < y$
LessThanOrEqual	Boolean	x, y:Numeric	returns $x <= y$
FloatFromInt	Float	i:Integer	returns i as a Float
ProbFromFloat	Probability	f:Float	convert f to a probability in the range $[0, 1]$
GraphAverageDegree	Float	None	returns graph average degree
AverageDegree	Float	l:NodeList	returns average degree of nodes in l
GraphMaxDegree	Integer	None	returns maximum degree of graph
MaxDegree	Integer	l:NodeList	returns maximum degree of nodes in l
GraphOrder	Integer	None	returns number of nodes in graph
GraphSize	Integer	None	returns number of edges in graph
TrueWithProb	Boolean	p:Probability	returns true with chance p, else false
NearestNeighbors	NodeList	d:Integer, l:NodeList	returns list of all nodes within d hops from nodes in l
IncidentEdges	EdgeList	l:NodeList	returns list of all edges with at least one endpoint in l
IncidentNodes	NodeList	l:EdgeList	returns a list of unique endpoints from edges in l
AllNodes	NodeList	None	returns list of all nodes in graph
AllEdges	EdgeList	None	returns list of all edges in graph
NodeListIntersection	NodeList	$l1, l2$:NodeList	returns list intersection of NodeLists $l1$ and $l2$
EdgeListIntersection	EdgeList	$l1, l2$:EdgeList	returns list intersection of EdgeLists $l1$ and $l2$
NodeListUnion	NodeList	$l1, l2$:NodeList	returns list union of NodeLists $l1$ and $l2$
EdgeListUnion	EdgeList	$l1, l2$:EdgeList	returns list union of EdgeLists $l1$ and $l2$
ListFilterWithProb	List	l:List, p:Probability	returns sublist of l randomly filtered with chance p
ListPortion	List	l:List, p:Probability	returns first $floor(p * length(l))$ elements of l
ListShuffle	List	l:List	returns elements of l in randomized order
GPConstantNode	List Numeric Boolean	None None None	returns an empty list returns randomly initialized number returns randomly initialized boolean

Table 6.2: Parameters

Parameter	Value
Population size	50
Evaluation limit	10000
Crossover chance	0.5
Mutation chance	0.5
Initial tree depth	2-7
Mutation tree depth	1-3
Evaluation test cases	10-30
Time steps per test case	100

The model parameter values used in these test cases are chosen to create noticeably different dynamic graph behavior while keeping the size of the graphs manageable computationally. With the exception of the final application, each of the following models was manually constructed using the available primitive set to ensure that the language was expressive enough to achieve the desired behavior. The final application investigates the use of this approach to model the dynamic behavior of an enterprise computer network.

6.5.1 Stable, Shrink, and Grow Models

The first three models use static values for the α parameter. The **stable** model uses an α of 0.0003 to create a model that adds and removes approximately the same number of edges at each time step as can be seen in the middle line in Figure 6.2. α values of 0.0001 and 0.0005 are used to define the **shrink** and **grow** models, respectively. These models correspond to the bottom and top lines in Figure 6.2.

6.5.2 Parameterized Model

The fourth application targets a **parameterized** version of the model during evolution. During evaluation, the evolved heuristics are tested against each of the models shown in Figure 6.2. To make this an achievable target, the model parameter values p, α, and β are made available to the evolved heuristics through additional terminal primitives: PInput, AlphaInput, and BetaInput, respectively. The evolutionary process must discover how to properly leverage this additional information.

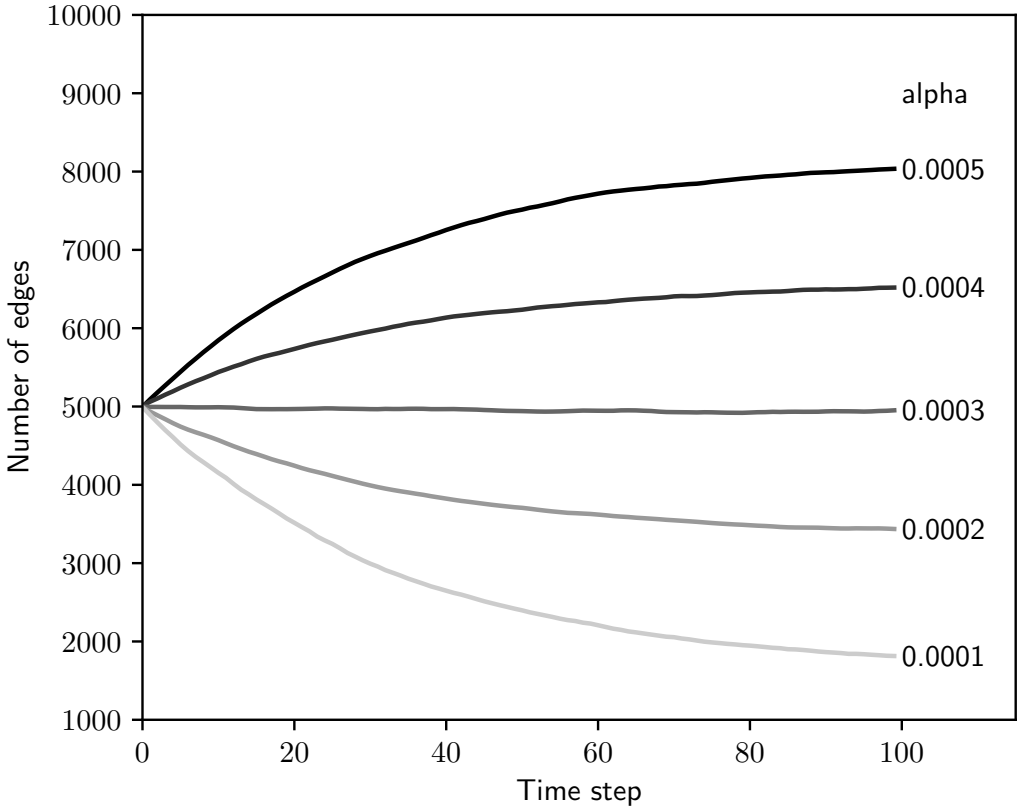


Figure 6.2: Graph size (number of edges) over time for various α settings. For each model, $n = 1000$, $p = 0.01$, and $\beta = 0.03$.

6.5.3 Changepoint Model

The fifth application model incorporates a sudden change in the model parameters. Initially, this **changepoint** model has an α value of 0.0001. At the halfway point of each evaluation, the α parameter is set to 0.0005 and the behavior of the dynamic graph changes noticeably. This trend is illustrated in Figure 6.3. Five additional primitives are added to provide the flexibility to handle this application. The TimeInput and TimeInputPercentage primitives return the time step and percentage of time steps passed, respectively. ChangepointInput returns the threshold time step (50) at which the model parameters change. In this work, the changepoint is simply provided to the evolved heuristics, but automated methods of detecting this transition exist [87]. ChangepointSwitch and ChangepointSwitchElse are conditional branching primitives that determine which branches to execute based on whether or not the changepoint has been reached.

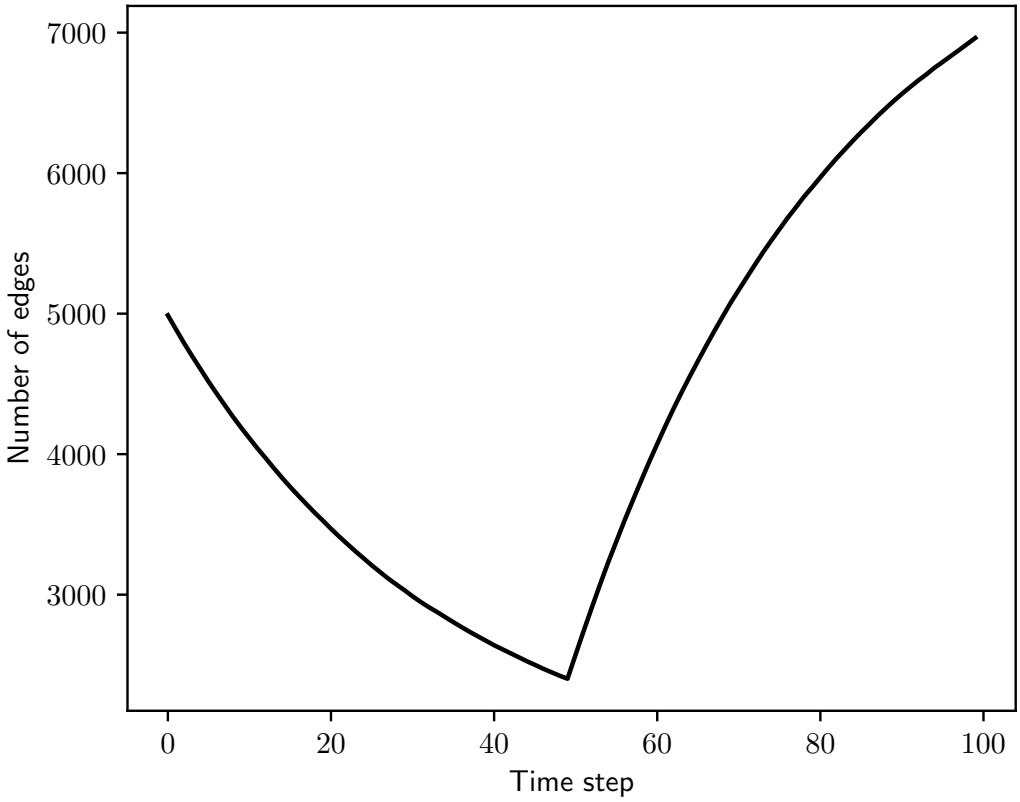


Figure 6.3: Average graph size over time for model with a parameter changepoint at time step 50.

6.5.4 Time-dependent Model

The final manually constructed application model includes a **time-dependent** model parameter. At each time step t, the α parameter's value is updated to $0.00001 * t$. The impact this has on the size of the graph can be seen in Figure 6.4. This application also leverages the TimeInput and TimeInputPercentage primitives described in Section 6.5.3.

6.5.5 Modeling Enterprise Network Traffic

The final application investigates the potential of this approach to model real-world phenomenon. NetFlow event logs are taken from the computer network at Los Alamos National Laboratory (LANL) [88] that contain information about communication sessions between pairs of computers on the network, such as the ports used or the amount of data transferred. A static graph

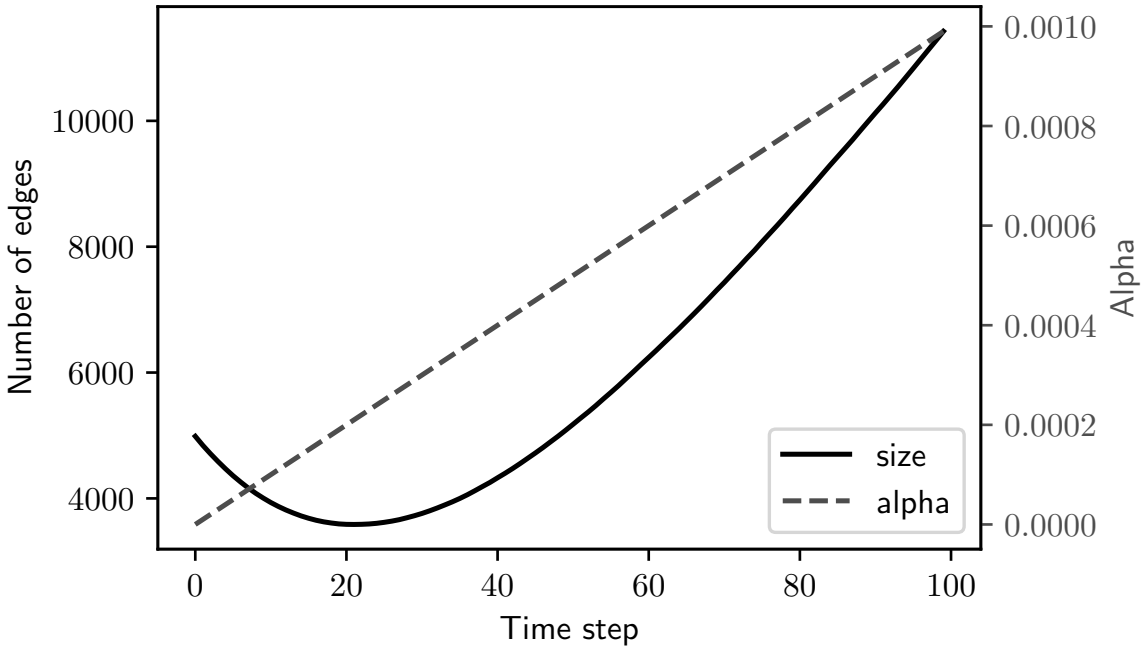


Figure 6.4: Average graph size over time for model with a time-dependent model parameter ($n = 1000$, $p = 0.01$, $\alpha(t) = 0.00001 * t$, $\beta = 0.03$). The dashed line indicates the value of the alpha parameter as it changes over time.

is generated for six minute increments during normal business hours (7am to 5pm) that contains an edge between two computer vertices if traffic is observed between those computers during that time window. To keep the evaluation time manageable for this proof-of-concept, the resulting graphs are reduced to activity between the most active 1000 computers. These static graphs are combined to produce a dynamic graph with 100 time steps for each of the 50 highest activity days. The down-selection in terms of days is done to remove non-business days, such as weekends and holidays, and provide a more consistent target for the evolutionary process to model. During solution evaluation, a subset of these days is chosen randomly without replacement to generate test cases. This application also leverages the TimeInput and TimeInputPercentage primitives described in Section 6.5.3.

6.6 Results and Discussion

Figure 6.5 shows the progression of fitness values over time during an evolutionary run targeting the NetFlow application. The top gray line indicates the best fitness seen so far during

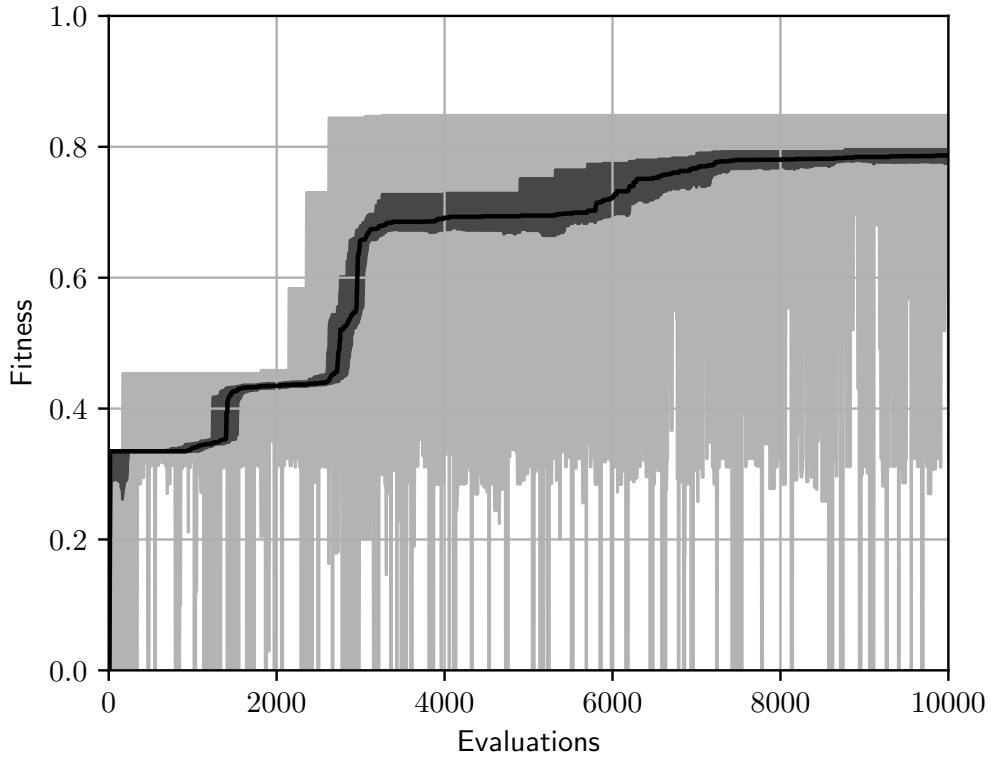


Figure 6.5: Fitness value over time for an example evolutionary run targeting the NetFlow application. The dark shaded region indicates the interquartile range of fitness values with the black line showing the median. The lighter shaded region shows the full range of population fitness values (minimum to maximum).

a run. The shaded region shows the interquartile range of population fitness values, with the black line indicating the median fitness. Although this is an example, it exhibits features seen in the results from other runs and applications. The large vertical jumps in the best fitness make it obvious where evolution has discovered a key functionality. These are typically followed by several smaller increases. Manual inspection reveals that the large jumps typically correspond to the introduction of entirely new subtrees whereas fine tuning of constant values lead to the smaller successive increases. Figure 6.6 shows an example summarization of the fitness trends for 30 experimental runs, this time for the parameterized application model.

Figure 6.8 summarizes the breakdown of the fitness scores of the highest-fitness solutions from 30 evolutionary runs for each application. The light shaded region shows the average objective values of the heuristics evolved for that application. Since these values are scaled to

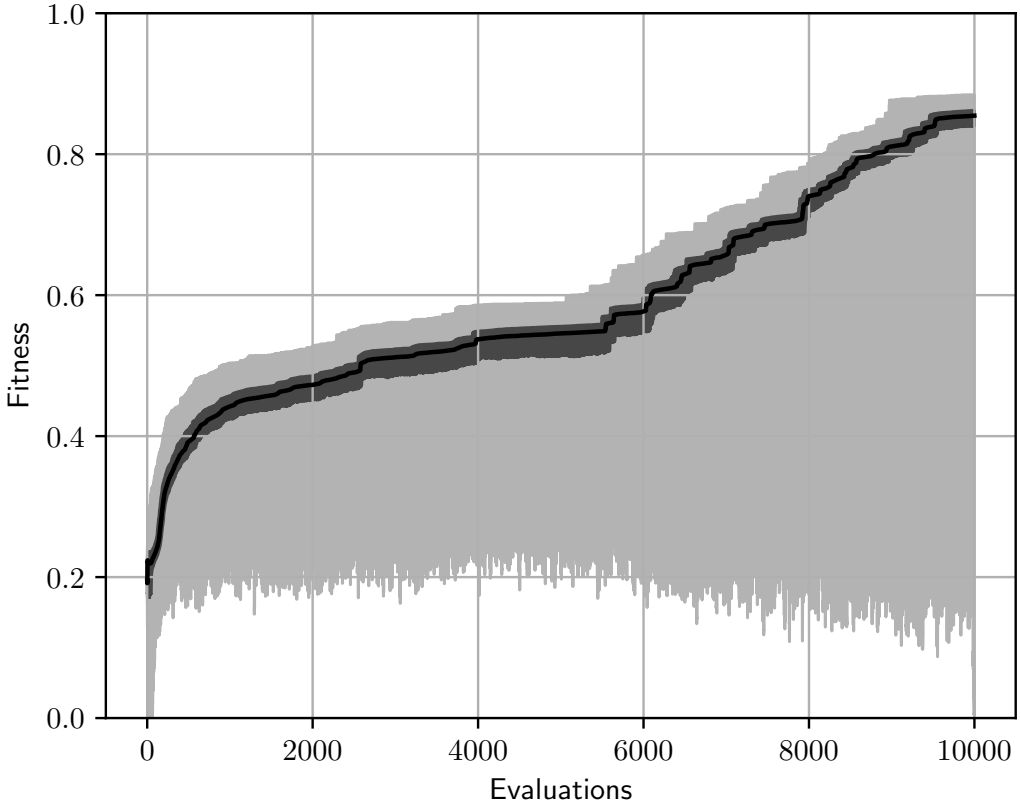


Figure 6.6: Population fitness values over time for the parameterized model application averaged over 30 evolutionary runs. The dark shaded region indicates the average interquartile range of fitness values with the black line showing the median. The lighter shaded region shows the average minimum to maximum range of population fitness values.

the objective values achieved by evaluating the model against itself, objective values closer to one indicate more accurate models.

The objective value results suggest that the size difference (SD) objective is the easiest to optimize. Note that due to the way objective values are scaled, a score under one can be the result of the model overfiting on the test cases during evaluation. Manual inspection reveals that overfitting occurs just as often as underfitting for each of the manually constructed target models. The NetFlow application, on the other hand, is more consistently underfit. This is likely the result of this application exhibiting far more unpredictable edge activity than the manually constructed models.

To illustrate the effect of the targeted evolution, the stable application model is also compared against each target model (except itself). The black region indicates the objective values achieved by this off-target model. Unsurprisingly, the stable model does poorly at mimicking the behavior of the other application models. However, this comparison demonstrates the need for measuring the multiple objective values. If, for instance, evaluation only considered the **SD** metric, this model would still perform relatively well on multiple applications despite obviously different behavior.

See Figure 6.7 for an example evolved parse tree targeting the parameterized application model. To visually analyze evolved parse trees, some rudimentary tree simplification techniques were automatically performed to reduce the size and complexity of the tree without changing the functionality. Examples include replacing arithmetic subtrees that produce constants with a single constant node or pruning operation subtrees that never actually change the graph. Note that this simplification is only done to help understand the behavior of the parse tree and is never used to modify the genotype of a solution during evolution. The example tree in Figure 6.7 was chosen for its relative simplicity. Many of the evolved solutions are still too large to include here even after simplification. To summarize this heuristic's behavior, it removes random edges with probability β, then adds random edges with probability α, and finally removes more random edges with probability $p * \beta$.

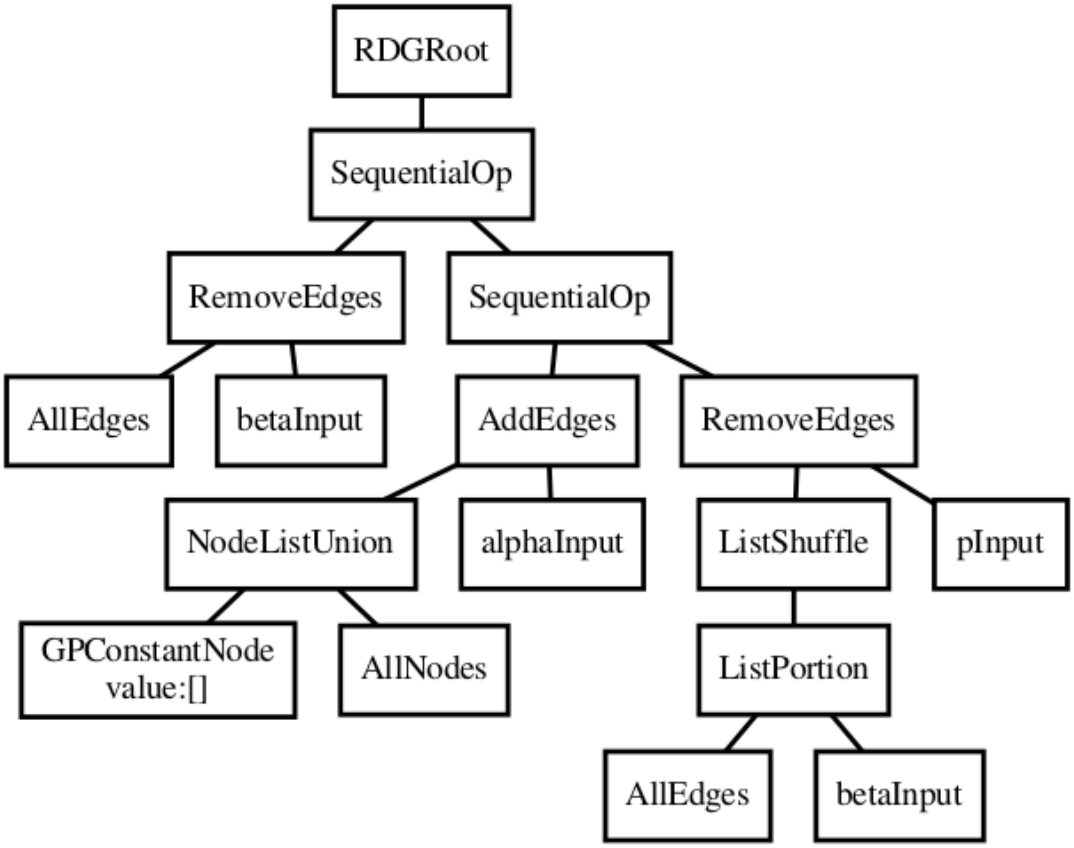
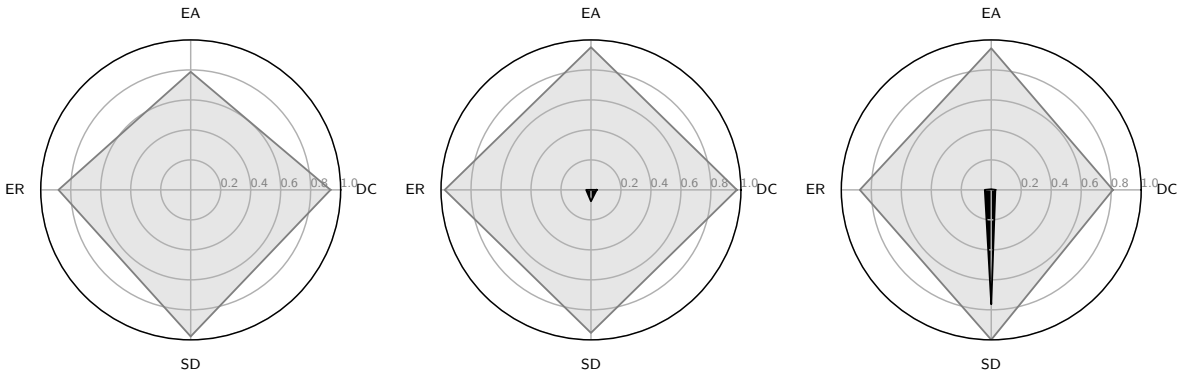
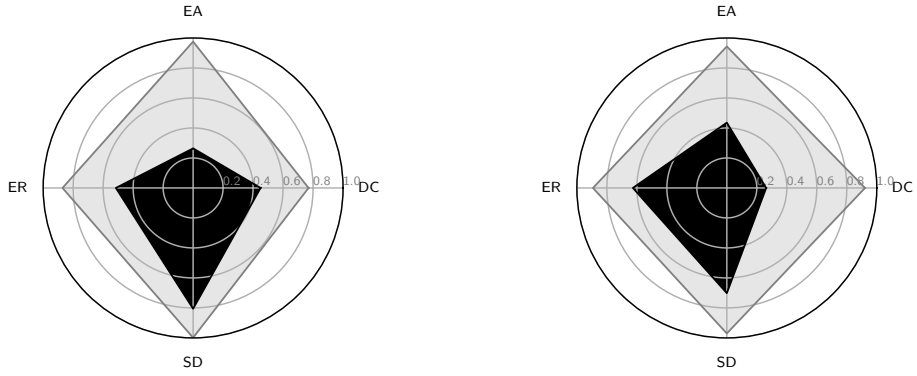


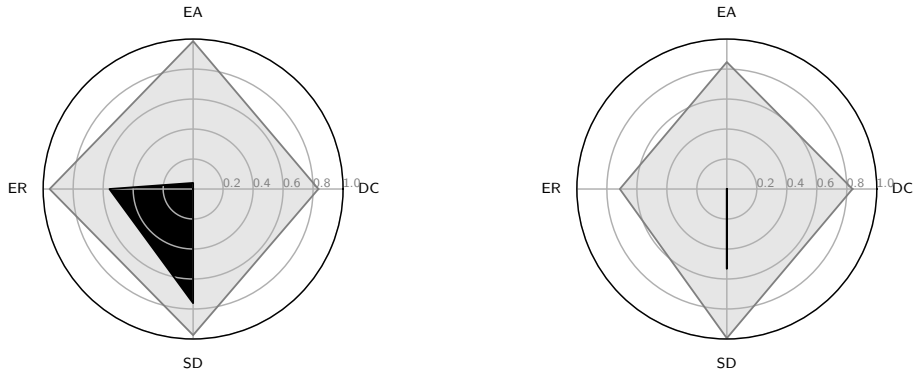
Figure 6.7: Example evolved parse tree targeting the parameterized application model. This parse tree has been simplified from its evolved form for clarity.



(a) Stable model (b) Shrink model (c) Grow model



(d) Parameterized model (e) Changepoint model



(f) Temporal model (g) NetFlow model

Figure 6.8: Objective values for each application model. The light shaded region shows the objective values achieved by the heuristic evolved for that application. For comparison, the black shaded region indicates the objective values measured for the stable model against each alternate model.

6.7 Conclusion

Random graph models are an invaluable tool in a variety of scientific domains. However, research in the field of random dynamic graph models is still relatively undeveloped. When modeling a dynamic concept with a random graph, the appropriate model must be selected for an accurate representation. Automated model selection techniques can be leveraged to identify the best choice from a pool of candidates, but this requires a versatile set of available models which is often limited when it comes to dynamic applications. Accurate models for new applications can be manually developed, but this process can be difficult and time-consuming.

This work investigated the potential of hyper-heuristics for automating the design of generative models for random dynamic graphs. Results demonstrate that the genetic programming approach has the capability to produce algorithms that accurately recreate the behavior of a number of random dynamic test models. Also, a preliminary proof-of-concept using enterprise network traffic data demonstrates the potential for leveraging this approach to model a variety of real-world concepts.

Chapter 7

Automated Design of Tailored Link Prediction Heuristics using Dynamic Primitive Granularity Control for Applications in Enterprise Network Security

The link prediction problem, which involves determining the likelihood of a relationship between objects, has numerous applications in the areas of recommendation systems, social networking, anomaly detection, and others. A variety of link prediction techniques have been developed to improve predictive performance for different application domains. Selection of the appropriate link prediction heuristic is critical which demonstrates the need for tailored solutions. This work explores the use of hyper-heuristics to automate the selection and generation of customized link prediction algorithms. A genetic programming approach is used to evolve novel solutions from functionality present in existing techniques that exploit characteristics of a specific application to improve performance. Final solution quality of this technique is further improved by leveraging a dynamic approach to controlling the level of the search granularity during heuristic optimization. Applications of this approach are tested using data from a real-world enterprise computer network to differentiate normal activity from randomly generated anomalous events. Results are presented that demonstrate the potential for the automated design of custom link prediction heuristics using dynamic granularity control that improve upon the predictive capabilities of conventional methods.

This chapter is an extension of the paper titled "Automated Design of Tailored Link Prediction Heuristics for Applications in Enterprise Network Security", which was published in the proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion [14]

7.1 Introduction

The link prediction problem involves predicting the existence of a relationship between entities. This problem occurs in a number of research domains and applications. Social networks have used link prediction to suggest new contacts [89]. Media streaming platforms leverage the technique to recommend material based on a customer's viewing history [90]. Link prediction techniques have also seen use within cybersecurity to differentiate anomalous behavior from normal activity [77]. This work leverages link prediction to identify abnormal activity in three network security applications.

Graphs provide a natural method of representing data about relationships between entities. In the social networking example, vertices in the graph represent network users and edges indicate connections between users. Utilizing a graph representation makes it possible to leverage graph theoretical approaches to network analysis. For example, social networks often use graph-based algorithms to detect communities within the network [91].

A common graph-based approach for predicting the existence of edges in a graph aims to position vertices in a space of latent (or hidden) features. Distances within this space are used to predict the likelihood of connections between the vertices. Information about the entities the vertices represent can be used to define these features. For example, a movie streaming application might ask a customer to choose their favorite movies or rank various genres to characterize the customer's interests.

However, many applications are restricted to the information contained in the topology of an existing network when making predictions for new links. A variety of techniques have been developed to predict new links based solely on existing edges (and the weights on those edges, if available) in a graph. Many of these techniques work by factorizing the graph's adjacency matrix. This approach produces a set of vectors that can be used as features of the vertices within the graph. Different methods of factorizing matrices have been used including singular-value decomposition [92], eigenvector decomposition [93], and Poisson matrix factorization [94].

The various link prediction methods differ in terms of complexity, efficiency, and predictive capabilities for a number of applications. The ongoing research developing new methods

demonstrates that the optimal technique likely depends on the specific application. Heuristic selection approaches can be used to identify the best algorithm for an application from a pool of candidate solutions, but this is limited by the quality and variety of available algorithms. New heuristics can be developed for additional applications, but this process can be difficult or expensive.

This work leverages a hyper-heuristic search to automate the design of novel link prediction heuristics that are tailored to improve predictive performance on specific problem applications. Unlike traditional optimization, which searches for an optimal solution to a specific problem instance, a hyper-heuristic instead searches the space of algorithms to find a heuristic that produces high-quality solutions to a class of problems. Hyper-heuristics can be selective, where they search for the best choice from a collection of available algorithms, or generative where they seek to create novel solutions from basic algorithmic building blocks.

For the generative hyper-heuristic approach employed in this work, functionality is extracted from existing techniques to create a set of primitive graph-based operations. Genetic programming (GP) is used to combine these operations to generate new heuristic solutions. The evolutionary search guides this process to optimize predictive performance for a specific problem.

Generative heuristic search techniques have been successfully utilized to construct and optimize algorithms in a number of application domains. Hyper-heuristic practitioners face a critical design decision when constructing the set of primitive operations that will be available to the heuristic search. Hypothetically, a Turing-complete set of operations should enable a heuristic search to find the optimal solution. However, this approach tasks the search with exploring the space of all possible computer programs making it unlikely that the search will produce acceptable solutions in a feasible amount of time. Search efficiency can be improved by leveraging problem domain knowledge to include high-level operations specifically targeted at the application. This can dramatically reduce the search time needed to find adequate solutions, but also has the potential to bias the search and restrict the flexibility needed to find the optimal solution.

The heuristic search used in this work incorporates a novel technique that allows leveraging domain knowledge to improve search efficiency without overly restricting the algorithmic flexibility need to express the optimal solution. Primitive operations are implemented at multiple abstraction levels and the abstraction level available to hyper-heuristic is dynamically altered during the search. A few methods of automatically controlling this primitive granularity level are investigated and their relative performance is compared. Results show that not only is the heuristic search leveraged in this work capable of improving performance for multiple real-world link prediction applications, but a dynamic approach to controlling the level of primitive granularity can result in additional performance gains.

7.2 Background

This section covers background information on graph representation and the link prediction problem, including some commonly used link prediction techniques. Also included is some background on evolutionary computation and genetic programming, which are leveraged in this work.

7.2.1 Graphs and Adjacency Matrices

A graph, $G(V, E)$, is made up of a set of vertices V and a set of edges between these vertices $E \subseteq V \times V$. If vertices $v_i, v_j \in V$ are connected, an edge (v_i, v_j) exists in E. The edge information can also be expressed as an adjacency matrix with dimensions $|V| \times |V|$. For each vertex $v_i \in V$, the i^{th} row and column of A described the outgoing and incoming edges of v_i. For vertices v_i and v_j, the value at $A[i][j]$ will be zero if no edge connects the vertices, or non-zero otherwise. For an undirected graph, A is symmetric as all incoming edges are also outgoing. If the graph does not contain self-loops, the diagonal entries of A are all zero. For graphs with weighted edges (as seen in this work), a non-zero value at $A[i][j]$ indicates the weight of the edge (v_i, v_j).

7.2.2 Link Prediction

The link prediction problem involves predicting the presence of a relationship between two entities. In graph terms, the goal is to determine the likelihood of an edge existing between any two vertices. This is typically done by leveraging historical topological information, known features of the entities, or some combination of the two.

A variety of link prediction techniques only consider local information that can be observed within the immediate neighborhood of the potential link endpoints. Node Popularity is possibly the simplest example and is calculated using only the relative popularity, or tendency to connect, of the endpoints. Conventionally, this is calculated using the formula $1 - exp(-2p_i p_j)$ where p_i and p_j are the popularity values of vertices v_i and v_j, respectively [95]. For simple undirected graphs, a vertex's degree can be used as an approximation for its popularity value.

There are numerous other methods that only utilize local information, including Common Neighbors, the Salton Index, the Jaccard Index, and the Preferential Attachment Index. For an overview of these methods, as well as several others, see [96]. Local link prediction methods are typically quick and easy to compute. Despite this, they can still be fairly accurate for some applications.

7.2.3 Adjacency Matrix Decomposition

The adjacency matrix representing a graph can be mathematically decomposed into two or more new matrices. For a normal decomposition, the decomposed matrices can be used to reconstruct the original adjacency matrix, often by repeated matrix multiplication. For example, singular value decomposition (SVD) decomposes a matrix $A \in \mathbb{R}^{n \times n}$ into the orthogonal matrices $U, V \in \mathbb{R}^{n \times n}$ and the diagonal matrix $D \in \mathbb{R}^{n \times n}$, where $A = UDV$. The n rows of U and columns of V can be used as feature vectors to characterize the vertices in the graph.

Making predictions with the matrices produced by normal decomposition is difficult because the matrices only capture the exact information about existing connections in the graph. Predictions based on these inputs will perfectly reproduce the original graph's connections.

Fortunately, methods exist that do a better job of predicting unseen edges by finding approximations for the decomposed matrices. For instance, the rank r truncated singular value decomposition instead produces the matrices $U_r \in \mathbb{R}^{n \times r}$, $V_r \in \mathbb{R}^{r \times n}$, and the diagonal matrix $D_r \in \mathbb{R}^{r \times r}$. For values of r strictly less than n, this is an approximate decomposition and $A \approx A_r = U_r D_r V_r$. The closer the value of r is to n, the more accurate the approximation A_r is to A.

While this approximate decomposition might seem counterintuitive, it has a couple of significant benefits. First, for large values of n and small values of r, storing U_r, V_r, and D_r requires less space than the original A. Second, the approximate nature of A_r makes it possible to inform predictions about edges that were not originally in the graph. Multiple techniques leverage this approximation approach for the link prediction task. Two examples that are used in this work are truncated singular value decomposition (TSVD) and truncated eigenvector decomposition (TED).

Both TSVD and TED produce matrices whose rows or columns can be used to generate length r feature vectors that characterize each of the graph's vertices. Conventionally, these are not used directly; the decomposed matrices produced are multiplied to obtain the approximate adjacency matrix A_r. The relative values of A_r are used to inform predictions. A high value at $A_r[i][j]$ suggests the edge (v_i, v_j) is likely. However, this work also considers using the feature vectors more directly by making them available as inputs to neural network classifiers.

7.2.4 Neural Network Classification

Neural networks have been used to classify inputs as normal or anomalous in a variety of applications [97, 98, 99]. In this work, neural network functionality can be included in evolved link prediction heuristics. These neural networks are tasked with producing a score for each input link that indicates how likely that link is to occur. The neural networks utilized in this work are relatively simple fully connected feed-forward networks with a variable number of levels.

To provide the neural network with useful information, they are capable of considering multiple input methods. The first is problem inputs, such as labels on the link endpoints. The

second allows the neural network to consider the output of other link prediction methods. Finally, neural networks can also consider the intermediate feature vectors produced by adjacency matrix decomposition techniques. The input methods used by each neural network, along with the architecture of these networks, is optimized by the hyper-heuristic search.

7.2.5 Evolutionary Computation

Evolutionary Algorithms (EAs) are a family of biologically inspired generate-and-test black-box search techniques [20]. Solutions are generated and assigned a fitness value that represents how well they solve the problem at hand. Initial solutions are typically generated randomly, but solutions in further generations are generally constructed by applying variation operators, such as crossover or mutation, to existing parent solutions. Some degree of selection pressure is conventionally applied to favor solutions with better fitness values when selecting parents or determining which solutions survive to future generations. The process continues until some termination criteria is met, such as a limit on execution time or convergence of solution fitness values.

7.2.6 Genetic Programming

Genetic Programming (GP) is an evolutionary search technique where the solutions take the form of executable programs. GP is also considered a form of hyper-heuristic search, where the goal is not to find the optimal solution to a specific problem but to find an algorithm that produces high quality solutions to a specific class of problems [7]. In this work, GP is used to evolve programs that perform the link prediction task. By guiding the evolution with input from a subclass of the link prediction problem, the GP finds tailored heuristics that exploit characteristics of that problem class to improve predictive performance.

These programs can be represented in a number of ways, but a common approach represents solutions as Koza-style parse trees [57]. Variation operators have been developed that act on this representation, such as subtree mutation, where a randomly selected subtree is replaced with a new randomly generated subtree, or subtree crossover, where two parent solutions exchange randomly selected subtrees. Parse tree solutions are made up of internal function nodes

and terminal leaf nodes. The set of operations and values used as nodes in these parse trees are referred to as primitives and depend on the application.

7.2.7 Primitive Granularity Control

In a conventional GP application, the set of primitive operations available to the search is decided a priori and does not change over the course of evolution. The construction of the primitive set has the potential to bias the search and have a significant impact on the performance of the GP. Typically, these primitive operations are extracted from existing algorithmic approaches to the target application. Even when the ideal set of operations is known, these primitives can be implemented at various levels of operation abstraction or granularity.

Practitioners can include complex primitives that have some key functionality that is targeted at the application in question. A set of such high-level operations can allow a GP to quickly find complex solutions that perform well. Unfortunately, these complex operations typically come in an "all or nothing" form. If an optimal solution requires a small modification to the provided functionality, the high-level primitive set might prevent the necessary fine-tuning.

Alternatively, a more granular set of primitives with lower level functionality can result in a GP with a far greater range of algorithmic expression. For instance, an abstract *for-loop* primitive can instead be decomposed into a more basic set of variable manipulation, conditional checking, and control flow branching operations. These lower level operations could be recombined to produce the functionality of other conditional branching abstractions such as the *do-while* and *switch-case* constructs. However, this improved flexibility can come at the cost of a dramatically increased search complexity as the GP must "reinvent the wheel" to achieve more complex functionality.

Previous work investigated the benefit of a technique known as dynamic primitive granularity control that aims to leverage the benefit of both the high-level and low-level approaches [100]. The primitive operation set is implemented at multiple levels of granularity and the granularity level available to the heuristic search can vary throughout evolution. Although the previous work demonstrated the potential gains of dynamically altering the level of primitive granularity,

it did this with an exhaustive search of granularity schedules which is infeasible for more complex applications. This work leverages and builds on the dynamic primitive granularity control technique by investigating multiple approaches to controlling the granularity level during the search.

7.3 Related Work

The link prediction problem has seen a lot of recent research activity. Both Wang et. al. [101] and Liben-Nowell et. al. [92] covered a variety of link prediction methods and applications in the field of social networking. Lü et. al. [96] summarized a number of approaches for other complex network types, such as those seen in biology and e-commerce. Many link prediction methods specifically leverage matrix decomposition. Dunlavy et. al. [102] discusses multiple decomposition-based techniques for temporal link prediction. Poisson matrix factorization [94] has seen a number of successful applications for link prediction, including in the field of cybersecurity [77]. The approach presented in this work leverages many of these methods as primitive operations available to the heuristic search. Unlike some of the application-specific link prediction methods discussed, the hyper-heuristic framework developed in this work is not limited to the applications presented here and can be easily applied to new problem domains.

The heuristic search utilized in this work is capable of combining multiple link prediction techniques to improve predictive performance. A number of ensemble learning methods take a similar approach. Gomes et. al. [103] describes a wide variety of ensemble learning techniques. The hyper-heuristic in this work is not limited to a preset method of combining multiple link prediction algorithms. The evolutionary search has the capability to optimize novel and often unintuitive ways to combine these algorithms to improve performance.

This work makes use of simple neural networks to classify links as likely or unlikely. Neural networks have a long history of being applied to such tasks [98]. More recently, a lot of activity has been seen in the field of neuroevolution which leverages evolutionary optimization to improve neural network performance [104]. More specifically, genetic programming has been applied to optimize neural network architectures [105]. This work uses a similar approach

to optimize neural networks for the link prediction task, but is also capable of combining these networks with alternative link prediction techniques for further performance gains.

Previous work demonstrated that increasing primitive granularity can improve the quality of final evolved heuristics at the cost of increased search time [60, 10]. This work employs dynamic granularity control in an attempt to achieve the benefits of the more flexible primitive set while mitigating against the increased execution time. Similar work has demonstrated the benefit of dynamically altering other evolutionary parameters, such as population size or number of offspring, that would conventionally be static [106]. The dynamic configurations found showed improvements in fitness when given an equivalent amount of time to run.

The process of constructing primitives with lower-level operations used in this work resembles attempts to leverage primitive modularity to allow GP to reuse complex functionality. Examples include Automatically Defined Functions [107], Evolutionary Module Acquisition [108], Hierarchy Locally Defined Modules [109], and Adaptive Representation [110]. The goal of these approaches is to make it easier for offspring to inherit more complex functionality that has been found useful in previous generations. This work, on the other hand, leverages problem-specific knowledge to provide high-level operations a priori that can be deconstructed during the evolutionary search to achieve greater search granularity.

7.4 Methodology

Genetic Programming (GP) is used to evolve a population of link prediction heuristics that are targeted at a specific application.

7.4.1 Initialization

Before the heuristic search is initiated, the link data is segmented into three partitions of configurable size. The first partition is the historical data that is used to inform the link prediction heuristics. All of the primitives described in Section 7.4.5 produce scorings for the input links based on this historical information. This includes neural network classifier components, which are trained on this data. To save execution time during the heuristic search, the fundamental

graph-based topological metrics are pre-computed whenever possible and stored for later access by the evolved heuristics.

The set of links to be scored is split to create the second and third partition of the link data. Fitness evaluation of evolved solutions is performed with the second partition as validation to determine the individual's ability to classify previously unseen links. The third partition is used as a hold-out to evaluate the final evolved solutions on data that was never seen during the search. This is done to ensure that the evolved solutions do not have inflated fitness values as a result of over-fitting on the historical and validation data.

7.4.2 Representation

Strongly typed parse trees [61] are used to represent evolved solutions. The initial pool of solutions is randomly generated from the available primitive operation set (described in Section 7.4.5) using a ramped half-and-half approach. An example parse tree representation of a basic link prediction heuristic can be seen in Figure 7.1.

7.4.3 Evaluation

During evaluation, an evolved solution is used to score a set of input edges for one or more test cases. For each test case, the scoring is compared to the true labels for the edges using a receiver operating characteristic (ROC) curve. A ROC curve compares the true positive rate (TPR) with the false positive rate (FPR) at different classification thresholds. The area under the ROC curve (AUC) is a value in the range $[0, 1]$ and is maximized when the scores for positive edge samples are consistently higher than the scores for negative edge samples. The fitness of an evolved solution is the average of the AUC values across each of the evaluation test cases. This fitness value is maximized when the link prediction heuristic clearly differentiates likely edges from unlikely ones.

If a heuristic contains a neural network subtree, the network is trained to classify links from the historical link data partition (the first partition described in Section 7.4.1). A configurable portion of the historical data is omitted from the training and used as validation in order to detect and prevent over-fitting of the model. This training takes place over a configurable number of

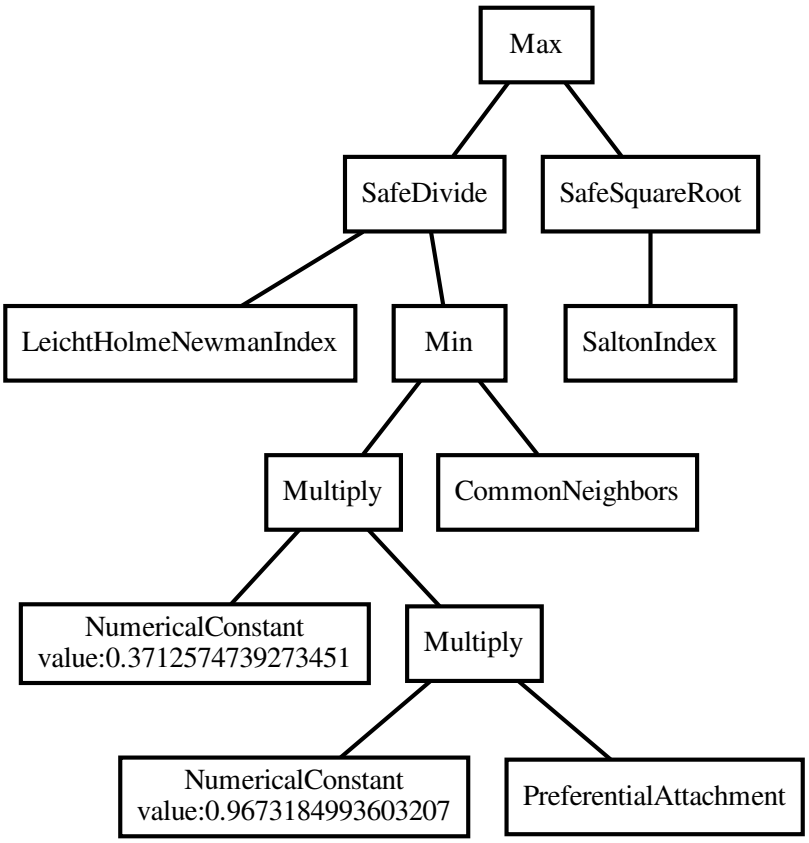


Figure 7.1: Example link prediction parse tree.

epochs. Neural networks are optimized using the Adam optimizer [111] with a binary cross-entropy loss function. If an offspring inherits a neural network model from one of its parents without the architecture of the network being altered during variation, the model parameters are also copied from the parent and the training process is skipped. This is done to prevent the wasting of unnecessary execution time repeatedly retraining the same neural networks.

7.4.4 Evolution

Parents are chosen using tournament selection. According to a configurable probability, either subtree crossover between two parents or subtree mutation from a single parent is used to generate new offspring solutions. Only crossover or mutation is used for a single offspring, not both, due to the dramatic effect of subtree crossover on a solution's genotype. If an offspring

Table 7.1: Primitive Operation Types

Type	**Description**
Integer	Returns a whole number
Float	Returns a floating point number
Weight	Returns a floating point number bound to the range $[0, 1]$
ScoreArray	Array of floating point values for each edge in the input list
NNInputLayer	Neural network input layer
NNHiddenLayer	Neural network hidden layer
NNDropoutLayer	Neural network dropout layer
NNActivationLayer	Neural network activation function layer
NNRegularizer	Kernel regularizer for neural network layer
NNInputVector	Feature vector formatted for input to a neural network
Matrix	Matrix of float values

inherits a neural network subtree from its parent and the subtree's architecture is not altered during variation, the state of the neural network is copied from the parent as well; this duplication reduces unnecessary computation time spent retraining identical networks. Offspring are added to the existing population, then truncation based on fitness is used for survival selection. If a configurable number of generations (convergence threshold) pass without seeing improvement in the population's best fitness, execution is terminated early.

7.4.5 Primitive Operations

As this work employs a strongly typed GP approach, each instance of an operation has an associated type to enforce compatibility. The available primitive types can be seen in Table 7.1. See Table 7.2 for a description of the basic set of primitive operations. These fundamental low-level operations are inspired by existing link prediction techniques as well as neural network and ensemble classifiers.

7.4.6 Dynamic Primitive Granularity Control

This work aims to improve upon previous work [14] by leveraging a dynamic approach to controlling the level of primitive operation granularity. The heuristic search has the ability to change the set of available primitive operations during evolution. The current granularity

Table 7.2: Basic Primitive Operations

Functions	
Add(X, Y)	element-wise vector addition
Subtract(X, Y)	element-wise vector subtraction
Multiply(f, X)	$f \times X$ element-wise scaling
Multiply(X, Y)	element-wise vector multiplication
SafeDivide(X, Y)	element-wise safe division (0 if $Y_i = 0$ else X_i/Y_i
SafeLog(X)	element-wise safe natural log (0 if $X_i \leq 0$ else $ln(X_i)$)
Exp(X)	element-wise exponential
Mean($X, Y, \dots$)	$(X + Y + \dots)/\lvert\{X, Y, \dots\}\rvert$
Absolute(X)	element-wise absolute value
SafeSquareRoot(X)	element-wise safe square root (0 if $X_i < 0$ else $\sqrt{X_i}$)
Min($X, Y, \dots$)	element-wise minimum
Max($X, Y, \dots$)	element-wise maximum
Rescale(X)	rescales values in X to the range $[0, 1]$
RankConvert(X)	converts values in X to evenly-spaced values in $[0, 1]$
MatrixMultiply(U, D, V)	scores based on decomposition ($U \times D \times V$)
NeuralNetworkRoot	root of a neural network subtree
Sigmoid	sigmoid activation function
RELU	rectified linear unit activation function
Dropout	dropout layer
Dense	typical densely connected network layer
L1Regularizer	L1 kernel regularizer
L2Regularizer	L2 kernel regularizer
InputLayer	input layer of neural network
Terminals	
UDegree(u, v)	weighted degree of u
VDegree(u, v)	weighted degree of v
CommonNeighbors(u, v)	number of length 2 (or 3 for bipartite graphs) paths between u and v
TotalNeighbors(u, v)	$\lvert$neighbors(u) $\cup$ neighbors(v)$\rvert$
TSVDU(A)	TSVD left feature matrix (U)
TSVDD(A)	TSVD diagonal matrix (D)
TSVDV(A)	TSVD right feature matrix (V)
TEDU(A)	TED left feature matrix (U)
TEDD(A)	TED diagonal matrix (D)
TEDV(A)	TED right feature matrix (V)
NumericalConstant	randomly initialized numerical constant

level determines the set of primitives available to the search when generating initial solutions or using variation to produce offspring.

In order to allow the primitive granularity level to change during the heuristic search, a second set of higher-level operations is also constructed. These macro operations are defined in terms of the basic operation set so that they can be automatically decomposed when the primitive granularity level is lowered. For instance, Figure 7.2 shows the parse tree in Figure 7.1 after the macro operations have been decomposed. See Table 7.3 for a description of the set of high level macro primitives used in this work. See Lü and Zhou [96] for descriptions of the various macro primitive terminal operations.

The process of constructing higher level primitives from lower level primitives can be repeated, creating a hierarchy of granularity levels as was done in previous work [100]. However, this work only employs two granularity levels, namely a low level and a high level. Investigation of the impact of additional granularity levels is left for future work.

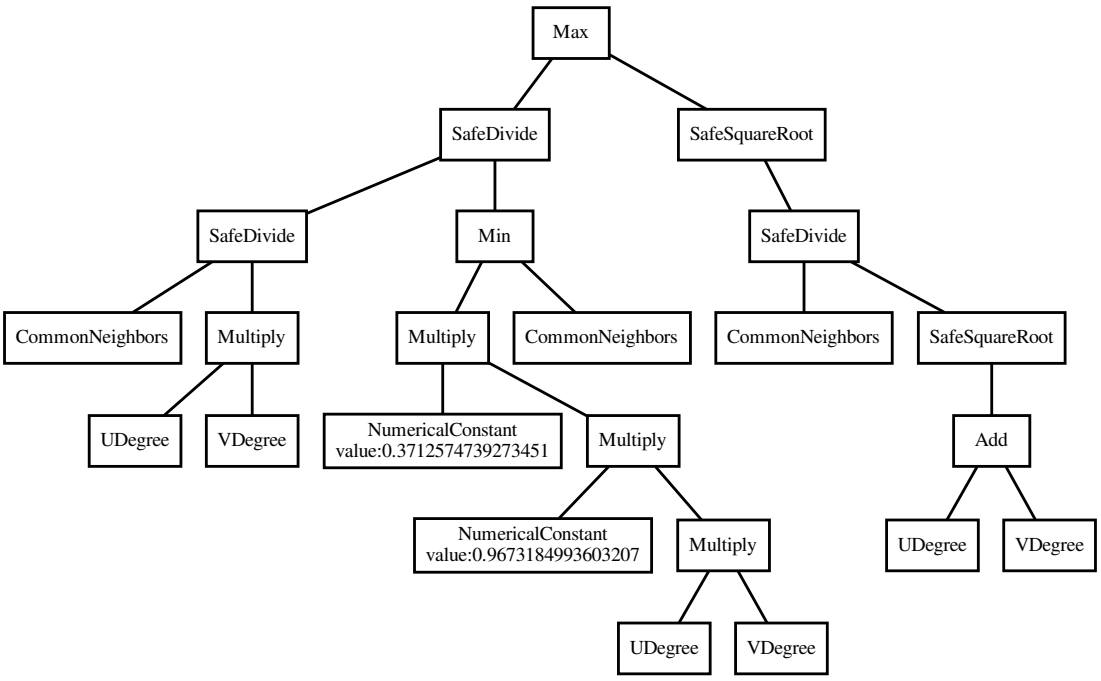


Figure 7.2: Parse tree shown in Figure 7.1 after having all macro primitives decomposed to lower the primitive granularity level.

Table 7.3: Macro Primitives

Functions	
ScaledMean($f_X, X, f_Y, Y, \dots$)	Mean(Multiply(f_X, X), Multiply(f_Y, Y) $\dots$)
RankMerge($X, Y, \dots$)	Mean(RankConvert(X), RankConvert(Y), $\dots$)
ScaledRankMerge($f_x, X, f_Y, Y, \dots$)	Mean(Multiply(f_X, RankConvert(X)), Multiply(f_Y, RankConvert(Y)), $\dots$)
SigmoidL1DenseLayer	Sigmoid(L1Regularizer(Dense))
SigmoidL2DenseLayer	Sigmoid(L2Regularizer(Dense))
SigmoidDropoutDenseLayer	Sigmoid(Dropout(Dense))
RELUL1DenseLayer	RELU(L1Regularizer(Dense))
RELUL2DenseLayer	RELU(L2Regularizer(Dense))
RELUDropoutDenseLayer	RELU(Dropout(Dense))
Terminals	
TSVDScore(A)	MatrixMultiply(TSVDU(A), TSVDD(A), TSVDV(A))
TEDScore(A)	MatrixMultiply(TEDU(A), TEDD(A), TEDV(A))
NodePopularity(u, v)	Subtract(1, Exp(Multiply(-2, Multiply(UDegree(u, v), VDegree(u, v)))))
SaltonIndex(u, v)	SafeDivide(CommonNeighbors(u, v), SafeSquareRoot(Add(UDegree(u, v), VDegree(u, v))))
JaccardIndex(u, v)	SafeDivide(CommonNeighbors(u, v), TotalNeighbors(u, v))
SorensonIndex(u, v)	SafeDivide(Multiply(2, CommonNeighbors(u, v)), Add(UDegree(u, v), VDegree(u, v)))
HubPromotedIndex(u, v)	SafeDivide(CommonNeighbors(u, v), Min(UDegree(u, v), VDegree(u, v)))
HubDepressedIndex(u, v)	SafeDivide(CommonNeighbors(u, v), Max(UDegree(u, v), VDegree(u, v)))
LeichtHolmeNewmanIndex(u, v)	SafeDivide(CommonNeighbors(u, v), Multiply(UDegree(u, v), VDegree(u, v)))
PreferentialAttachment(u, v)	Multiply(UDegree(u, v), VDegree(u, v))
TSVDInput(A)	InputLayer(TSVDU(A), TSVDD(A), TSVDV(A))
TEDInput(A)	InputLayer(TEDU(A), TEDD(A), TEDV(A))

Multiple methods of controlling the granularity level are explored in this work. See Table 7.4 for a summary of these control schemes. The simplest configurations, which are labeled **StaticLow** and **StaticHigh**, keep the granularity level constant throughout the search. These runs correspond to the conventional GP approach with an operation set decided a priori. Preliminary experimentation also investigated using the combined set of high and low-level primitives simultaneously. However, the results from these early trials were poor, possibly a result of the dramatic increase in the search complexity due to the very large primitive set. Results from these trials are not included here, but could warrant further investigation in future work.

The next two methods start at one level, then switch to the other at the midway point during the search. These configurations are labeled **LowToHigh** and **HighToLow**, depending on their initial setting. The **Alternating** method switches the granularity level whenever a configurable number of evaluations pass without any improvement in the best fitness value seen. Lack of improvement in the population's average fitness could also be used as a condition for the **Alternating** method, but this was not used for this work because it would trigger significantly less often and make it harder to investigate the impact of changing the primitive granularity level.

The final **SelfAdaptive** method encodes the primitive granularity level into the genotype of the individual solutions. This value is randomly determined during population initialization. During recombination, potential parent pairs are selected with matching granularity levels to reduce the mixing of low and high-level primitives. Offspring inherit the primitive level of their parents. During mutation, the granularity level associated with an individual has a configurable chance of alternating.

If the primitive level of the search (or the individual's primitive level in the **SelfAdaptive** scheme) changes from high to low, the macro operations within the solution are automatically decomposed before variation operators are applied. Automatic reconstruction of macro primitives when the granularity level changes from low to high is not currently implemented, but this is possible and could be explored in future work. It should also be noted that due to the strongly-typed parse tree representation, it is not always possible to construct a solution entirely using high-level primitives. In these cases, the tree generation process can fall back to low-level

operations when no viable high-level operation is available for the necessary primitive type; this can occur during initial parse tree generation or subtree mutation.

Table 7.4: Primitive Granularity Level Control Schemes

StaticLow	low throughout evolution
StaticHigh	high throughout evolution
LowToHigh	low initially, change to high at midpoint
HighToLow	high initially, change to low at midpoint
Alternating	random initially, alternate on convergence
SelfAdaptive	self-adaptive granularity level

7.4.7 Parameters

See Table 7.5 for a list of the configurable parameter values used in this work. Most of these values were inspired by previous work automating the development of link prediction heuristics [14]. The execution time limit was chosen to ensure convergence in the majority of evolutionary runs. More extensive tuning targeting a specific application would likely improve the quality of the final solutions. However, the results presented in this work demonstrate that even without application-specific tuning, the GP is able to produce heuristics that outperform tuned versions of the conventional methods.

7.5 Experiment

To demonstrate the potential of this approach to improve predictive performance for real-world applications, this work is applied to multiple network security prediction tasks. These applications utilize data collected from the enterprise computer network at Los Alamos National Laboratory (LANL) [88]. The data set contains traffic information in the form of NetFlow entries as well as networked host logs that track authentication and process execution events. Evolved heuristics are evaluated by how well they differentiate legitimate network activity from randomly generated anomalous events.

Table 7.5: Parameters

Parameter	**Value**
Population size	80
Offspring size	40
Execution time limit	6 hours
Convergence threshold for **Alternating** method	100 evaluations
Crossover chance	0.7
Mutation chance	0.3
Granularity level mutation chance for **SelfAdaptive** method	0.1
Parent selection tournament size	5
Initial minimum tree depth	1
Initial maximum tree depth	5
Mutation minimum tree depth	1
Mutation maximum tree depth	3
Neural network validation split	0.6
Neural network training epochs	20

7.5.1 Predicting Process Execution

Process execution events are collected from the LANL data to create two types of graphs. The first contains vertices for user accounts and process names, along with edges that indicate a process was executed by (or on the behalf of) a user. The second replaces the users with computers connected to processes that have been executed on those computers. Edge weights indicate the number of times a user-process or computer-process pair was seen in the data. It is worth noting that these process execution graphs are bipartite; the vertex set can be divided into the set of users (or computers) and the set of processes and edges can only connect a vertex in one set to a vertex in the opposite set. Only links that connect a user (or computer) and a process are considered for prediction.

7.5.2 Predicting Network Traffic

A graph is constructed to represent the communication between devices on the LANL network. An edge indicates that the two devices it connects communicated at least once. Edges are weighted by the number of distinct communication sessions occurring between devices in the

data set. Unlike the process execution application, these network traffic graphs are not bipartite. Any pairing of two networked devices is considered a valid link during prediction.

7.5.3 Training and Evaluation

The first four weeks (28 days) of data are used to generate the initial historical graph for each application. See Table 7.6 for a summary of the data set used. Adjacency matrices are created for each of these graphs using the transformation $A[i][j] = ln(1 + weight(v_i, v_j))$. The logarithmic transformation can improve accuracy when very active links have weights that are orders of magnitude greater than those of low activity links. This bursty activity behavior is common in many real-world applications including those targeted in this work.

Table 7.6: Data Set Summary

User-Process	
Unique users	25,761
Unique processes	27,944
Historical user-process links	2,106,120
Total user-process test links	216,352
Computer-Process	
Unique computers	13,465
Unique processes	27,944
Historical computer-process links	1,976,705
Total computer-process test links	190,857
Network Traffic	
Unique devices	60,185
Historical communication links	1,136,854
Total communication test links	250,815

Each of the following 14 days is used to create an evaluation link prediction test case. Links seen on these days are compared to the historical graphs. To be included in the test case, both endpoints must be present in the historical graph, but the link itself must not be present. This requires that the evolved heuristics be able to predict new (previously unseen) links, but does not expect the solutions to be able to predict links to vertices they have no historical information about.

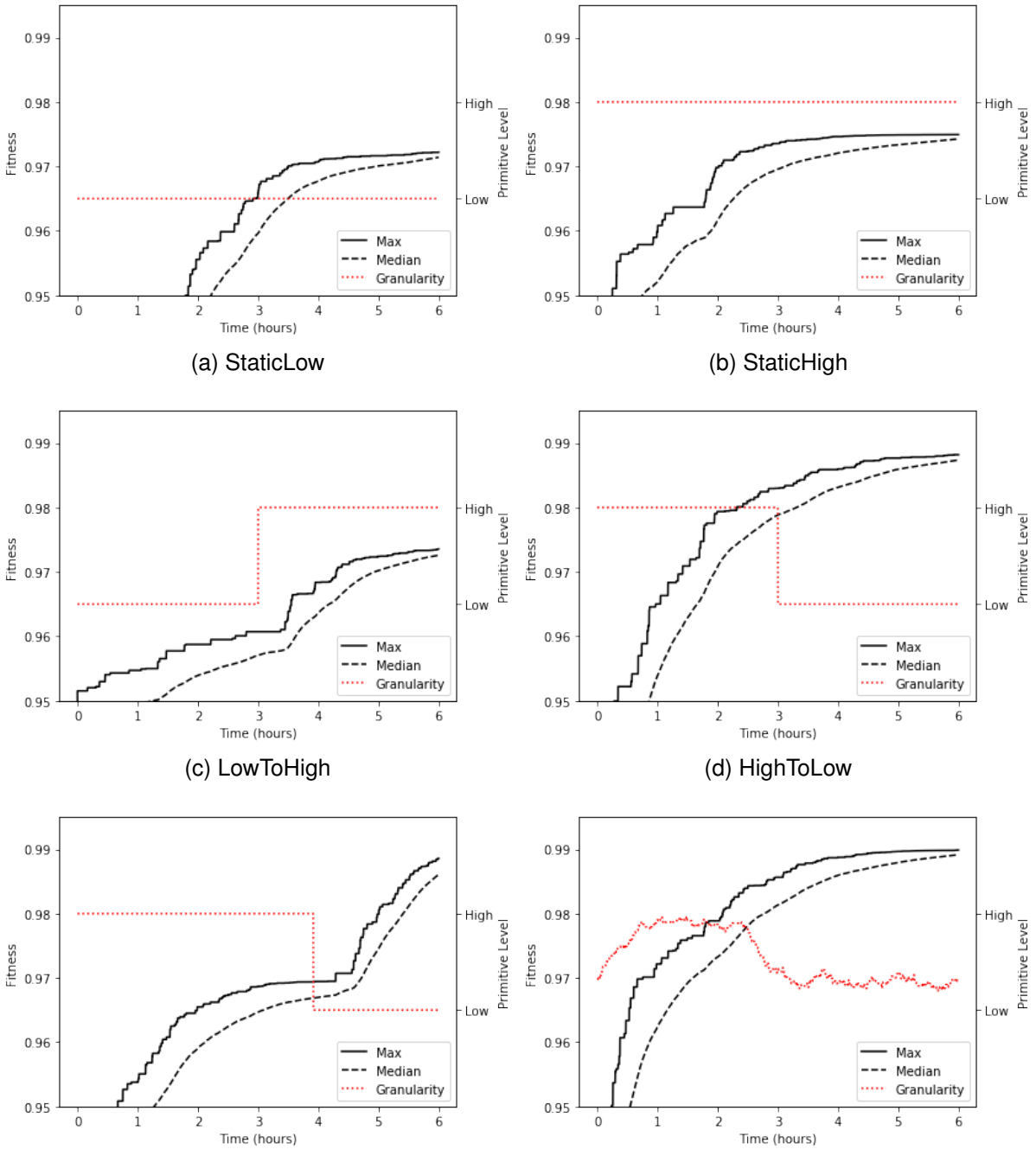
The collections of new links are labeled as positive samples for each test case. To provide negative samples, an equal number of links missing from both the historical graph and the test case are randomly selected. The positive and negative samples are concatenated for each test case and the order of the samples is randomized.

For each application and each dynamic granularity level control scheme described in Section 7.4.6, a population of heuristics is evolved with the goal of differentiating the positive and negative samples from each test case. Evolved candidate solutions are executed for each test case to produce a score for each sample. These scores are compared to the true labels to produce an AUC score for each test case and the solution's fitness is its average AUC.

To examine the benefit of heuristic specialization, the best evolved solution from each application is also applied to each of the other applications and compared to the heuristic specifically evolved for that application. Since the heuristic search is capable of combining multiple link prediction methods to improve predictive performance, an ensemble classification method is also used for comparison [103]. This classifier calculates scores using a weighted average of the scaled output of each of the basic link prediction methods seen in Table 7.3. The weights used for this average are tuned for each application individually using a random-restart stochastic hill climbing search. Preliminary experimentation investigated alternative methods for this tuning, such as an evolutionary algorithm, but each of the methods produced similar results.

7.6 Results and Discussion

Figure 7.3 shows a visualization of the fitness values over time during an example evolutionary run for each primitive granularity control scheme targeting the User-Process application. Although these are selected examples, they have been chosen because they are representative of the behavior commonly seen in repeated searches. The axes on each figure is kept the same to enable easier comparison and focus on the smaller fitness increases found near the end of each run. Also shown is the granularity level at each point during the search. For all but the SelfAdaptive control scheme, this level is simply at the high or low level at any given point. For the SelfAdaptive scheme, the value shown ranges between high and low to indicate the percentage of the population at each level over the course of the search.



(a) StaticLow (b) StaticHigh

(c) LowToHigh (d) HighToLow

(e) Alternating (f) SelfAdaptive

Figure 7.3: Population fitness values versus execution time from example evolutionary runs for each configuration targeting the User-Process application. The best fitness seen so far during the search is shown by the solid black line and the dashed black line shows the population median fitness. The dashed red line indicates the primitive granularity as set by the indicated control scheme.

Searches that begin at the high primitive level tend to reach much higher fitness values earlier in the search. This is not surprising since the higher level macro operations provide hand-crafted link prediction techniques to the search. However, these searches also tend to converge quite early as well.

In the **StaticLow** configuration, fitness gains are slow and the search often does not converge in the allotted time. In fact, preliminary experimentation shows that these searches will often continue to find gradual increases if allowed to run much longer. However, even with extra search time, the heuristics evolved under the **StaticLow** control scheme perform poorly when compared to any run that incorporates the high level macro primitives.

Despite the difficulty of finding high-quality solutions with the low level primitive set alone, results show that searching at the high level, then moving to the low level can improve final solution quality significantly. This is especialy true for the example shown for the **Alternating** control scheme. In this case, moving to the low primitive level allowed the search to escape what appears to be a local optima and continue to find improvements.

The example shown for the **SelfAdaptive** scheme shows the evolutionary search exploiting the benefits of starting at a high level and transitioning to the low level later in the search. Solutions in the initial population that are randomly generated at the high level tend to have much higher fitness values than those at the low level. This disparity causes high-level solutions to proliferate and the overall population primitive level rises. Later in the search, transitioning to the low primitive level enables more fine-tuning of the evolved heuristics and prevents premature convergence.

The best fitness seen over time for each application and primitive control scheme, averaged over five evolutionary runs can be seen in Figure 7.4. Even when the results are averaged over repeated runs, several of the fitness trends show obvious changepoints that correspond to transitions between the primitive granularity levels. The trends in fitness growth vary between applications, suggesting that the optimal primitive level control scheme likely depends on the target application.

For the NetFlow application, the fitness values produced by the final solutions from the **StaticLow**, **StaticHigh**, and **LowToHigh** configurations were substantially lower than the

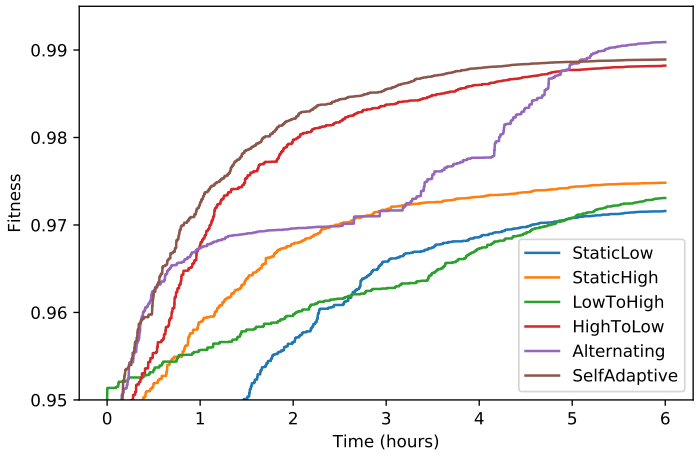
other control schemes. This is likely the result of the NetFlow application being more complex than the process execution applications. This application has fewer historical links to base prediction on and has more links to predict than both of the process execution applications. Additionally, there is an order of magnitude increase in the number of potential links due to the higher number of possible endpoints and the fact that network communication sessions are not bipartite as the computer-process and user-process graphs are.

For all three applications, the **StaticLow** and **StaticHigh** strategies produce relatively poor results while the **Alternating** and **SelfAdaptive** control schemes tend to perform better. These results provide further evidence of the strength of dynamically controlling the primitive granularity level during evolution. The fact that the **HighToLow** control scheme also consistently outperforms the **LowToHigh** scheme suggests that, at least for these applications, there is more to gain by starting with more complex primitive operations initially, then refining the search by moving to a lower primitive level.

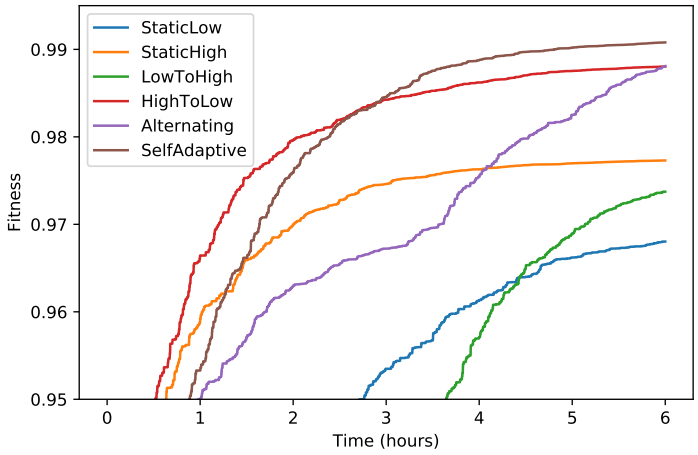
To compare the various link prediction methods on each application, the held-out test case links are combined into a single link prediction task. The ensemble classifier tuned for each application is used on these test cases, along with the best evolved heuristic produced by each of the primitive level control schemes. Additionally, heuristics evolved to target the other applications are also evaluated and the best performing solution is included for comparison.

Figure 7.5 shows the receiver operating characteristic (ROC) curve that result from the link scorings produced by each method for each application. The range of these plots are restricted to the top left corner to allow easier comparison of the various ROC curves. Curves closer to the top left correspond to more accurate link prediction methods. The targeted ensemble classifiers perform relatively well, beating out several of the evolved heuristics. However, for each application, there are multiple evolved solutions that improve upon the ensemble classifier's predictive performance.

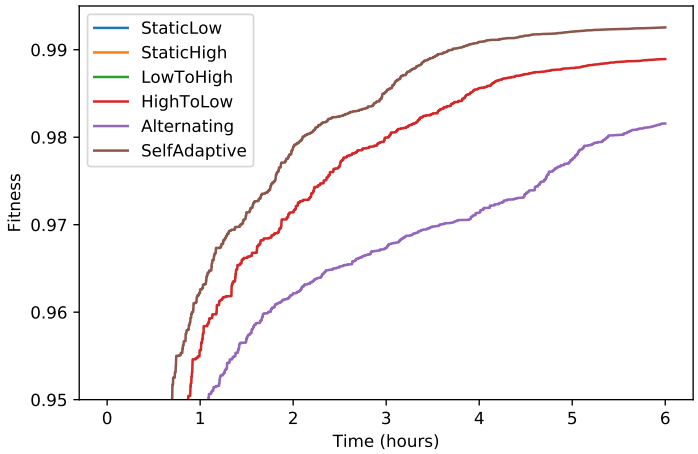
The genetic programming approach to generating tailored link prediction heuristics is capable of more flexibility than the traditional ensemble method. Manual inspection of some of these evolved heuristics reveals that performance is often improved by combining the basic



(a) User-Process

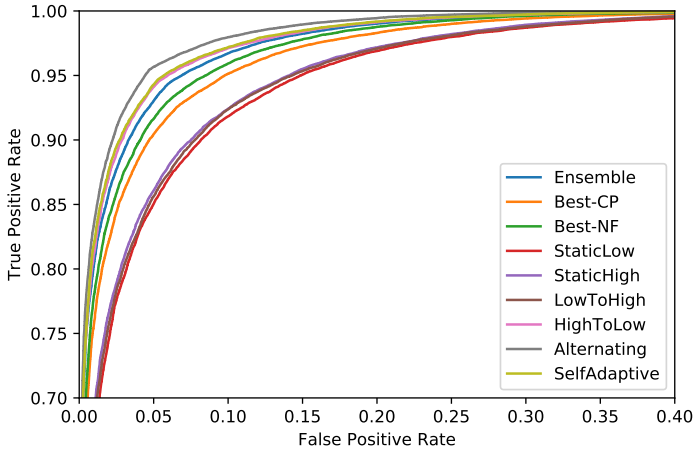


(b) Computer-Process

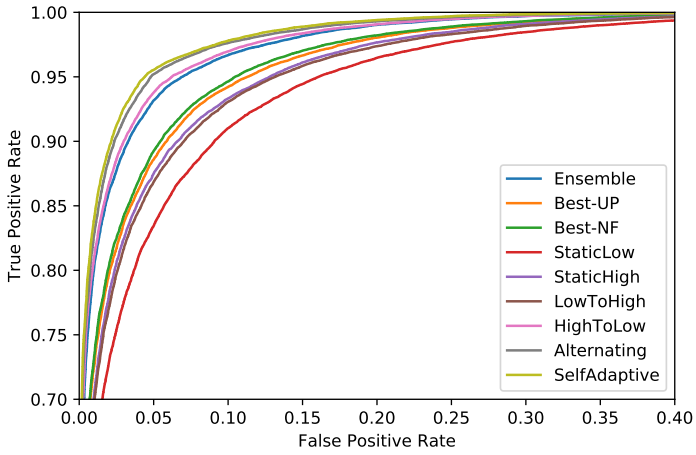


(c) NetFlow

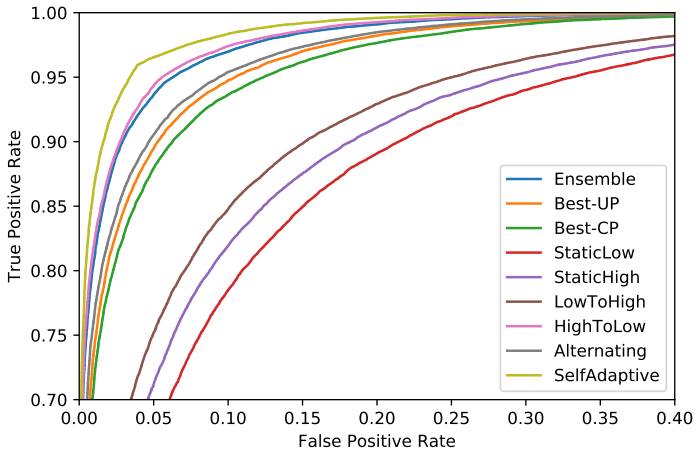
Figure 7.4: Best fitness values over time for each primitive level control scheme. Results are averaged over five evolutionary runs. Fitness ranges shown are set to be consistent with those in Figure 7.3 for easy comparison. The missing schemes in the plot for the NetFlow application produced fitness values too low to be seen.



(a) User-Process



(b) Computer-Process



(c) NetFlow

Figure 7.5: Comparison of receiver operating characteristic (ROC) curves produced by the best evolved heuristic using each primitive granularity control scheme for each application. The area of these plots is focused on the upper left corner to make it easier to see the differences between the curves. Curves closer to the upper left corner indicate better predictive performance.

Table 7.7: Link Prediction Accuracy

Method	Application		
	UP	**CP**	**NF**
Ensemble	0.98757	0.98734	0.9884
Best-UP	———	0.97995	0.98133
Best-CP	0.98277	———	0.97816
Best-NF	0.98518	0.98098	———
StaticLow	0.97269	0.97005	0.9296
StaticHigh	0.975	0.97748	0.94082
LowToHigh	0.97428	0.97625	0.95065
HighToLow	0.98863	0.98835	0.9895
Alternating	**0.9911**	**0.99019**	0.98343
SelfAdaptive	0.98906	**0.99106**	**0.99285**

AUCs produced by the method indicated on the left for the User-Process (**UP**), Computer-Process (**CP**), and NetFlow (**NF**) applications. The rows labeled **Best-*** indicate the highest AUC achieved by applying a heuristic evolved to target a different application. The bold values indicate statistically better performance compared to other methods for each application according to t-test comparisons at the $\alpha = 0.05$ level. For the **CP** application, the **Alternating** and **SelfAdaptive** results were statistically indistinguishable.

link prediction methods in some unintuitive ways. For example, evolved heuristics often combine multiple configurations of the same basic link prediction method with different parameter values. The dynamic primitive granularity control further improves on this flexibility by allowing basic link prediction methods to be decomposed and further optimized for the target application.

Table 7.7 shows the numerical results of the ROC comparison in Figure 7.5. The values indicate the area under the ROC curves (AUC) for each compared method. Again, this comparison also includes the results of applying heuristics that are evolved to target different applications. In several cases, these off-target heuristics actually outperform the products of some of the targeted heuristic searches. This provides further evidence that selecting the proper primitive level control scheme can be critical as the wrong choice can dramatically reduce the final solution quality.

It should be noted that numerically, the gains in predictive performance can seem relatively minor when comparing link prediction methods. However, when using link prediction to identify anomalous computer and network activity, a small difference in the AUC can correspond to thousands (or more) of additional false positive results each day that could end up getting unnecessarily investigated. Even worse, this loss in performance could mean miss-classifying thousands of events associated with malicious activity as false negatives.

Manual inspection of the evolved solutions reveals some interesting trends. High-fitness individuals tend to be very complex, often containing over a hundred operations from the basic primitive set. This is especially true for solutions produced from runs that transitioned from the high to low level later in the search. The fact that the high-level primitive set allows for a more compact representation of complex functionality likely contributes to this as the size of the solutions multiplies when the macro primitives are decomposed. Although the higher fitness values suggest that this added complexity is being put to good use (as opposed to the trees simply being bloated with unneeded functionality), it does make it more difficult to get an intuitive understanding of how the more complicated heuristics work.

The vast majority of high-fitness solutions make use of multiple neural networks and these networks tend to have a significant impact on the final output of the evolved heuristics. However, almost no high-fitness solutions are seen that exclusively use neural networks to classify links. Of the existing link prediction methods listed in Table 7.3, TSVDScore and TEDScore tend to have higher impact on link scorings than the others although all of them appear in final solutions in some form. The ensemble link prediction classifiers trained for each application also favor these matrix decomposition techniques, although never exclusively.

Interestingly, the matrix decomposition methods tend to get used differently as neural network inputs than when they are used directly for their link scorings. The evolutionary search tends to rely on TVSDScore and TEDScore operations that leverage crude approximations of the adjacency matrix (truncation levels in the range $[10, 50]$) to produce link scores. Alternatively, neural networks often employ TVSDInput and TEDInput operations with truncation values over 200. The fact that TSVD and TED, when used directly, do better with lower truncation values is not surprising as this is often the case when they are used to predict previously

unseen links. However, the neural networks obviously benefit from the increased granularity provided by higher truncation values.

The neural networks produced by the evolutionary search tend to be relatively simple compared to more conventional machine learning applications such as image classification. The vast majority of these networks are only two or three layers deep and these layers typically have less than two or three hundred neurons. Instead of employing large, complex networks, evolved solutions tend to contain multiple simple networks.

7.7 Conclusion

Link prediction is a commonly occurring problem with applications in recommendation systems, social networking, anomaly detection, and others. Research in the development of new link prediction heuristics has produced a variety of techniques with various accuracy and efficiency trade-offs. Performance in link prediction applications requires leveraging the appropriate method. While the selection of the best link prediction heuristic can be automated, this relies on the quality and variety of available heuristics.

This work demonstrates the potential of using a generative hyper-heuristic search to automate the development of novel link prediction heuristics that are customized to specific applications. Results targeting three cybersecurity prediction tasks using real-world enterprise network data show that the evolutionary process is capable of improving predictive performance over baseline techniques by exploiting characteristics of the problem subclasses. This improved performance comes at the cost of additional a priori computation time, but the resulting solutions provide higher accuracy in applications where investigating false positives can be costly and time consuming.

This work also illustrates the benefits of leveraging a dynamic approach to controlling the level of primitive granularity throughout the evolutionary search. The investigated control schemes had significant impacts on the final evolved solution quality. However, these results also demonstrate that the optimal control scheme likely depends on the specific application. Further investigation into methods of automating the primitive level control process is warranted.

Chapter 8

Automated Design of Multi-Level Network Partitioning Heuristics Employing Self-Adaptive Primitive Granularity Control

Network segmentation has a variety of applications, including computer network security. A well segmented computer network is less likely to result in information leaks and more resilient to adversarial traversal. Conventionally network segmentation approaches rely on graph partitioning algorithms. However, general-purpose graph partitioning solutions are just that, general purpose. These approaches do not exploit specific topological characteristics present in certain classes of networks. Tailored partition methods can be developed to target specific domains, but this process can be time consuming and difficult. This work builds on previous research automating the development of customized graph partitioning heuristics by incorporating a dynamic approach to controlling the granularity of the heuristic search. The potential of this approach is demonstrated using two real-world complex network applications. Results show that the automated design process is capable of fine tuning graph partitioning heuristics that sacrifice generality for improved performance on specific networks.

8.1 Introduction

The network segmentation problem involves dividing a network into separate components. This can be done for a number of different reasons and on multiple types of networks. For instance, a computer network can be segmented into separate domains using traffic control rules to prevent malware from spreading across the network or to limit the traversal of an intruder. Strictly

enforcing best security practices when building a computer network can result in good segmentation, but this can get in the way of legitimate user productivity. This is especially true in the age of centralized login and IT management systems.

Graph partitioning methods have been applied to the network segmentation problem in a number of domains. This has the advantage of finding minimal cost partitions, which can be used to minimize the monetary cost of making changes to the network or reduce the impact on user productivity. The downside to this approach is that optimal graph partitioning is known to be NP-hard in general. Many graph partitioning applications are too complex and time sensitive to rely on computing an optimal partition. For these applications, practitioners often turn to heuristics that approximate the optimal graph partitioning within a reasonable amount of time.

One of the more popular approximate graph partitioning methods is known as multi-level graph partitioning. This approach involves approximating the input graph with a much simpler graph, partitioning this approximation, then using the partition to find an adequate partition to the original input graph. The phases of a multi-level partitioning algorithm are typically referred to as *coarsening*, *partitioning*, and *uncoarsening/refinement*.

Multi-level partitioning has been proven to be capable of quickly producing low-cost partition solutions for numerous complex real-world applications. Previous work has also shown that partition solution quality can be further improved by tailoring the graph partitioning heuristics used to the specific application. By targeting a subclass of graphs related to a certain application, the heuristic can exploit graph characteristics common to that domain. This tailoring process can be accomplished by manually designing new partitioning heuristics, but this process can be difficult and expensive.

Hyper-heuristic search techniques have been used to automate the design and optimization of novel graph partitioning heuristics that are customized to specific application domains. Instead of conventional optimization methods, which seek to find the optimal solution to a specific problem instance, hyper-heuristics search in the space of algorithms to find a program that produces high-quality solutions to a class of problems. A common hyper-heuristic approach

is genetic programming (GP), which uses a biologically inspired search process that evolves a population of executable programs.

This work builds upon previous research that leveraged GP to evolve tailored graph partitioning heuristics that improved on general-purpose off-the-shelf partition approximation methods. The previous search results are improved by incorporating functionality that allows the GP to dynamically alter the search granularity over the coarse of evolution. Results are demonstrated by evolving graph partitioning heuristics that are tailored to specific computer network security applications.

8.2 Background

This section reviews some background information on graphs and graph partitioning methods. Also covered is details on evolutionary and heuristic search techniques.

8.2.1 Graph Representation

Graphs are a powerful tool for representing concepts in which relationships are critical elements. A wide variety of applications lend themselves well to a graph-based representation, such as the connections between users on a social network or communication links between pairs of hosts on a computer network. A graph G is made up of a set of vertices V and a set of edges E. An edge connecting vertices $u, v \in V$ is denoted as (u, v). Edges can be directed, but the applications targeted in this work have undirected edges such that $(u, v) \in E \iff (v, u) \in E$.

Both edges and vertices can have associated weight values. Let W_V and W_E be functions that define the graph's vertex and edge weights, respectively. The weight of vertex u is denoted as $W_V(u)$ and the weight of edge (u, v) is $W_E(u, v)$. Let the total weight of a set of vertices X be $W_V(X) = \sum_{u \in X} W_V(u)$ and the total weight of a set of edges Y be $W_E(Y) = \sum_{(u,v) \in Y} W_E(u, v)$. Therefore, the total graph vertex weight is defined as $W_V(V)$ and the total graph edge weight is $W_E(E)$.

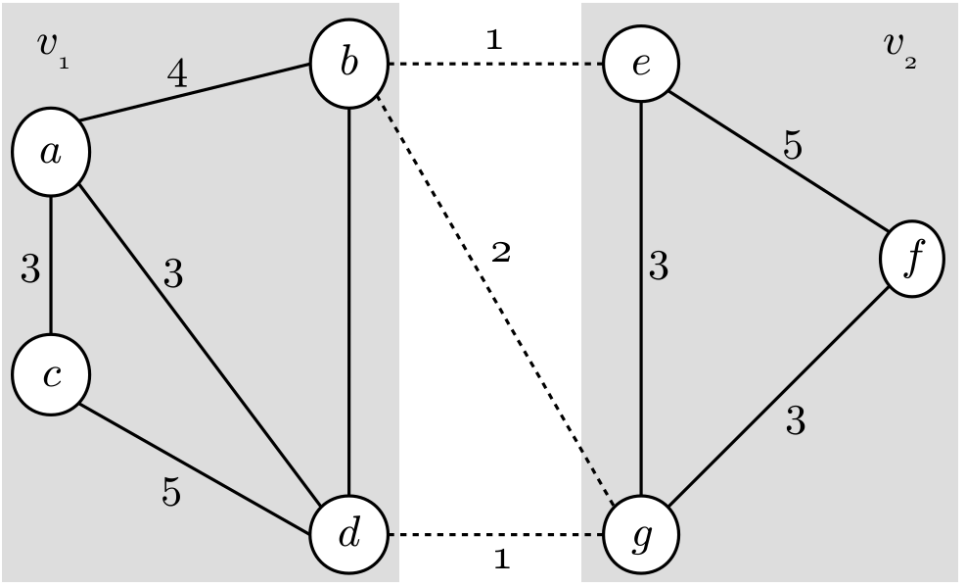


Figure 8.1: Example graph partition with vertex sets $V_1 = \{a, b, c, d\}$ and $V_2 = \{e, f, g\}$. Edges between partitions are indicated by dashed lines. The cut-cost of the partition is 4.

8.2.2 Graph Partitioning

Given a graph $G(V, E, W_V, W_E)$ and an integer k, a k-way partitioning of G divides the vertex set V into k disjoint subsets, $V_1, V_2, \ldots V_k$ such that $V_i \cap V_j = \emptyset$ if $i \neq j$. A *balanced* k-way partition requires that the ratio of the weight of the heaviest vertex subset to the average vertex subset weight not exceed $1+\epsilon$, where ϵ is the partition's *imbalance factor*. For a given partition, let P_E be the subset of edges in E that connect vertices in different vertex subsets. The total weight of this edge subset $W_E(P_E)$ is known as the *cost* or *cut-cost* of the partition. Many graph partitioning applications require that this cost be minimized as it represents the cost of modifying the input graph. Figure 8.1 shows an example graph partition.

8.2.3 Multi-level Graph Partitioning

Multi-level graph partitioning is an approximation method for finding a low-cost partitioning for an input graph. This process is broken into three distinct phases known as *coarsening*, *partitioning*, and *uncoarsening/refinement*. During the *coarsening* phase, the input graph is approximated using a sequence of smaller graphs. This is typically accomplished by contracting connected sets of vertices, replacing the subgraph with a single vertex.

Once the input graph is sufficiently compressed, which typically requires having fewer than some configurable number of vertices, a partition is computed for the smallest approximation graph during the *partition* phase. Because the approximated graph is much simpler than the original input graph, often several orders of magnitude fewer vertices and edges, an optimal partition can be computed quickly. However, multi-level partitioning has been shown capable of producing high-quality partitions even if the partition on the graph approximation is itself a rough approximation. For this reason, this partition is often found using very fast and simple stochastic techniques.

After a partition is found for the smallest approximation graph, it is used as a starting point for a partition in the next smallest graph in the series of graph approximations. Typically, some simple and efficient method of improving the partition, such as a local search that swaps vertices in different vertex subsets, is applied before moving to the next graph in the approximation series. This *uncoarsening* and *refinement* process is repeated until a partitioning is found for the original input graph. See Figure 8.2 for a visual overview of the entire multi-level partitioning approach.

8.2.4 Hyper-Heuristics

Traditional optimization techniques seek to find the optimal solution to a particular problem instance, such as the optimal partition to a specific input graph. Hyper-heuristics, on the other hand, search in the space of algorithms to find a heuristic that is optimized for a subclass of problems, such as a partitioning heuristic that produces low-cost partitions for a certain type of network. Genetic programming (GP) is a common hyper-heuristic search technique that uses an evolutionary algorithm (EA) to generate and optimize a population of executable programs.

In GP, as with other EAs, is initialized by generating (often randomly) a population of candidate solutions. Offspring solutions are created by inheriting genetic information from parents from previous generations. Variation operators, such as recombination and mutation, introduce changes to the inherited genes to encourage exploration of the solution space. Individuals in the population are evaluated against the target problem and assigned a *fitness* value that reflects

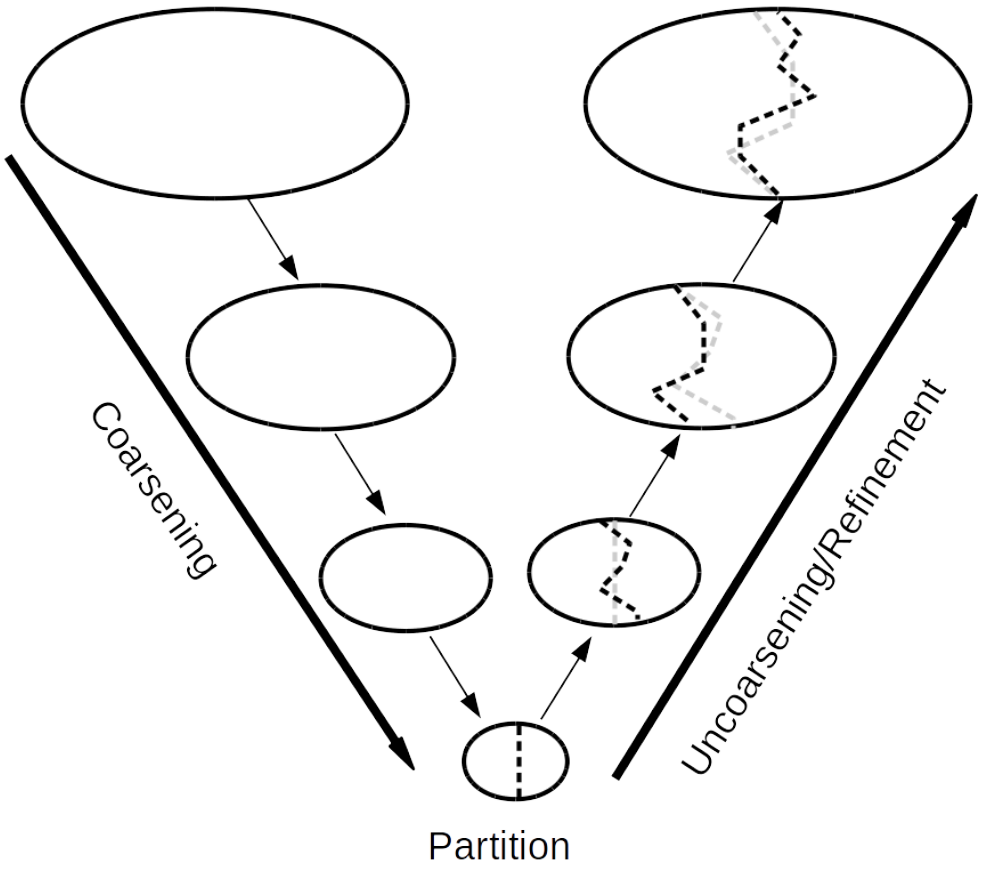


Figure 8.2: Multi-level graph partitioning strategy.

how well they solve the problem. When selecting individuals to pass on their genes or survive to future generations, individuals with higher fitness values are favored to apply selection pressure and drive evolution to exploit useful genetic information. The process of generating and evaluating offspring solutions repeats generation after generation until some termination criteria is met, such as a minimum fitness level or execution time limit.

GP differs from most other EAs in that individual solutions take the form of executable programs. There are multiple ways of representing program solutions that allow evolutionary variation operators. This work uses a common approach that encodes solutions as parse trees; an example graph partition heuristic parse tree can be seen in Figure 8.3. Subtree mutation and crossover operations are used to generate offspring from parent donors. Solutions are executed on input problems from a problem class and evaluated by the relative quality of their output.

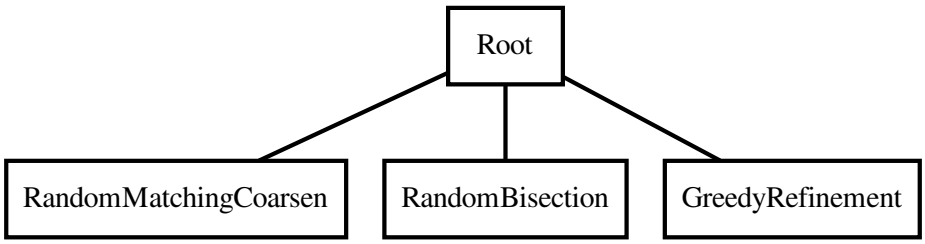


Figure 8.3: Example graph partition heuristic parse tree.

In this work, candidate graph partitioning heuristics are evaluated by the cost of the partitions they produce when used on a specific type of network.

8.2.5 Primitive Operation Granularity

Individuals in the population of a GP are constructed by combining algorithmic building blocks referred to as *primitive operations*. A set of these operations is typically collected by observing the functionality of related algorithms. However, this functionality can be extracted at differing levels of abstraction or complexity. Two GP practitioners might implement the same set of core functionality at radically different levels of operation granularity. This decision is typically made a priori and remains constant once the function set is implemented.

Previous work has shown that this choice can have a dramatic impact on the behavior of the heuristic search, affecting search efficiency as well as final solution quality. A set of high-level complex primitive operations can allow a GP to leverage domain expertise from hand-crafted approaches. Unfortunately, this can overly restrict algorithmic expressiveness as these complex operations are used in an "all or nothing" manner. Alternatively, a more fundamental low-level operation set can allow for more flexibility in the heuristic search. However, this often requires the GP "reinvent the wheel" to build up the complex functionality that might be required to solve a problem.

This work leverages a technique, known as dynamic primitive granularity control, that attempts to get the benefits of both of these options while mitigating their weaknesses. A flexible set of low-level functionalities is included in the primitive operation set. Additionally, a set

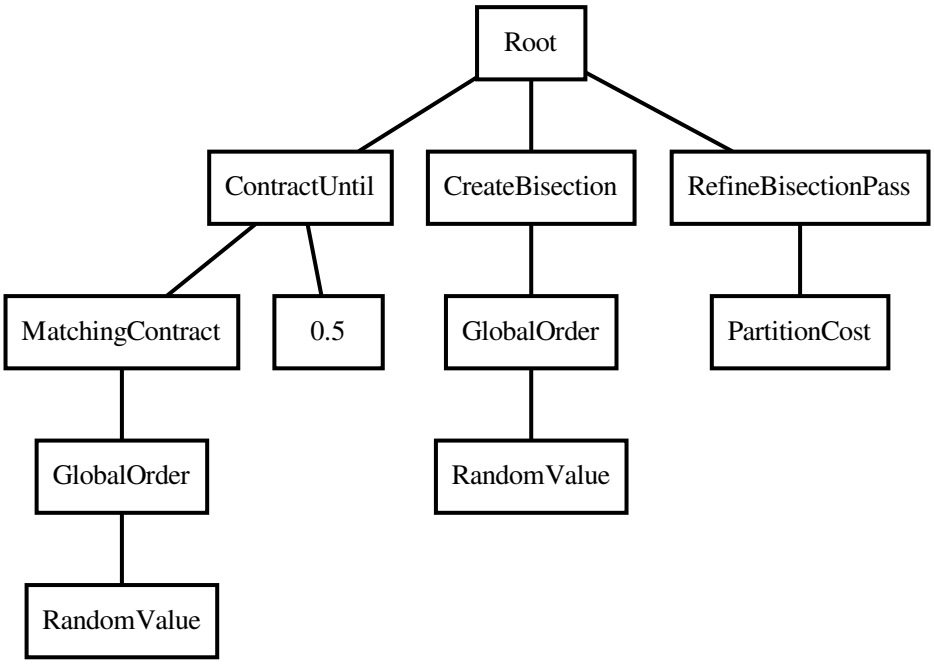


Figure 8.4: Decomposed version of the example graph partition heuristic parse tree shown in Figure 8.3.

of high-level complex operations is implemented by combining multiple low-level primitives. These "macro" operations can be decomposed into their more basic forms to allow fine tuning during the evolutionary search. For example, the decomposed version of the parse tree shown in Figure 8.3 can be seen in Figure 8.4. This allows the heuristic search to operate at multiple levels of primitive granularity. Previous work investigated a variety of methods for controlling this granularity level during the search. This work incorporates one of these methods which controls the level using a self-adaptive approach allowing the GP to optimize the granularity level while conducting the search.

8.3 Methodology

A genetic programming (GP) hyper-heuristic search is used to evolve a population of tailored heuristics to partition a specific class of graphs.

8.3.1 Representation

Individuals in the population are represented using strongly typed parse trees [61]. The typed representation allows more complex operation types while ensuring compatibility between operations.

8.3.2 Initialization

The initial population is generated randomly using a ramped half-and-half approach. Parse tree depth is controlled by configurable minimum and maximum settings. Under the self-adaptive primitive granularity control scheme, solutions are initialized with a randomly selected high or low primitive level designation. Solutions are constructed from the set of primitive operations available according to the appropriate level of primitive granularity.

8.3.3 Evaluation

Candidate solutions are executed on multiple input test cases taken from the target class of graphs. For each test case, the percentage of the edges *not* removed by the partition is calculated. A solution's fitness is the mean of these values across all test cases. The fitness value of a solution is given by

$$Fitness = \frac{1}{|T|} \sum_{t \in T} \left[1 - \frac{W_E(P_t)}{W_E(E_t)} \right],$$

where T is the set of test cases, P_t is the set of edges removed by the partition for test case t, and E_t is the original set of all edges for test case t. As a result of this transformation, fitness values scale from zero to one, with values closer to one corresponding to a heuristic that produces lower-cost partitions. If a solution takes an excessive amount of time to produce a partition, or the partition it produces is imbalanced, the individual is assigned a fitness value of zero. This penalty discourages solutions that are inefficient or do not maintain partition balance.

8.3.4 Evolution

Parents are chosen according to tournament selection. Offspring are generated using subtree mutation and subtree crossover. Due to the large genetic impact subtree crosover can have on

an individual, an offspring is only subjected to mutation or crossover, not both. With the self-adaptive primitive granularity approach, crossover matches parent pairs by their granularity level gene during crossover and the offspring inherits this level. During mutation, this gene has a random chance of being flipped. The current primitive level controls the set of operations that are used when generating a new subtree for subtree mutation. If an individual's primitive level mutates from high to low, all high-level primitive operations in the individual are decomposed to their low-level representations. Truncation is used for survival selection. To combat the high level of elitism introduced by truncation, parent selection tournament size is kept low to reduce selection pressure. Evolution is terminated by a configurable limit on search execution time.

8.3.5 Primitive Operations

The set of available primitive operations builds on previous work evolving multi-level graph partitioning algorithms [11]. A description of the low-level primitive operation set can be seen in Table 8.1. Table 8.2 lists the high-level operation set and describes how each of these macro operations is implemented in terms of the low-level primitives. Due to the strong typing used in this work, constructing a solution entirely of high-level operations is not always possible. In these cases, the parse tree generation and variation mechanisms can fall back to the low-level operation set when necessary.

8.3.6 Parameters

The search parameters used in this work can be seen in Table 8.3. These parameter values were inspired by previous work evolving graph partitioning heuristics [11]. Future work will include tuning these parameters for these specific applications.

8.4 Experiment

Tailored graph partitioning heuristics are evolved to target two real-world complex network segmentation applications. Data collected from the computer network at Los Alamos National Laboratory (LANL) is used to construct two types of graph-based models of network activity [88]. A summary of the details of the data used can be seen in Table 8.4.

Table 8.1: Low-level Primitive Operations

Primitive	**Description**
Root(G)	performs coarsening, partitioning, and refinement on G, returns partitioned graph
Add(x, y)	$x + y$
Subtract(x, y)	$x - y$
Multiply(x, y)	$x \times y$
SafeDivide(x, y)	0 if $y = 0$ else x/y
Negate(x)	$-x$
SafeInverse(x)	0 if $x = 0$ else $1/x$
SafeLog(x)	0 if $x \leq 0$ else $ln(x)$
Exp(x)	e^x
Min($x, y, \dots$)	minimum
Max($x, y, \dots$)	maximum
EdgeDegree(u, v)	$degree(u) + degree(v)$
EdgeWeight(u, v)	$W_E[(u, v)]$
EdgeNodeWeight(u, v)	$W_V(u) + W_V(v)$
NodeDegree(u)	$degree(u)$
NodeWeight(u)	$W_V(u)$
NodeEdgeWeight(u)	$\sum_{v \in neighbors(u)} W_E(u, v)$
MatchingContract(u, v)	contracts (u, v) if neither endpoint has been contracted this iteration
SubgraphContract(u, v, l)	contracts (u, v) if $(w_v[u] + w_v[v])/W_V(V) \leq l$
ContractUntil(r)	executes coarsening subtree until no change or contraction ratio r is exceeded
GlobalOrder(E, m)	order edges in E according to metric $m(u, v)$
LocalOrder(E, m)	order edges in E according to random walk, ranking incident edges according to metric $m(u, v)$
CreateBisection(V)	creates bisection of vertex set V according to input order
WhileImproves	repeats refinement until it stops improving bisection cost
RefineBisectionPass	attempts to swap vertex pairs in bisection to reduce partition cost
PartitionCost	returns cost of current bisection
NumericalConstant	randomly initialized numerical constant
RandomValue	return random value on each call

Table 8.2: High-level Primitive Operations

Primitive	**Description**
RandomMatchingCoarsen	ContractUntil(MatchingContract(GlobalOrder(RandomValue)), 0.5)
LightEdgeCoarsen	ContractUntil(MatchingContract(GlobalOrder(EdgeWeight)), 0.5)
HeavyEdgeCoarsen	ContractUntil(MatchingContract(GlobalOrder(Negate(EdgeWeight))), 0.5)
GloballyGreedyCoarsen	ContractUntil(SubgraphContract(GlobalOrder(m), l), r)
LocallyGreedyCoarsen	ContractUntil(SubgraphContract(LocalOrder(m), l), r)
RandomBisection	CreateBisection(GlobalOrder(RandomValue))
GraphGrowingPartition	CreateBisection(LocalOrder(RandomValue))
GreedyGraphGrowingPartition	CreateBisection(LocalOrder(PartitionCost))
KLBisection	WhileImproves(RefineBisectionPass(CreateBisection(GlobalOrder(RandomValue))))
GreedyRefinement	RefineBisectionPass(PartitionCost)
KLRefinement	WhileImproves(RefineBisectionPass(PartitionCost))

The first application models authentication activity between pairs of users and computers. This graph is bipartite, with two disjoint vertex sets, namely user account vertices and networked host vertices. An edge in this graph connects a user account vertex and a computer vertex and corresponds to a successful authentication in which the account credentials were used to access the computer. Edges are weighted by the number of distinct authentication events seen between each user and computer pair. An edge removal performed by a partition being applied to this network would correspond to revoking a user's ability to access a specific computer.

Graph partitioning has been used to segment networks of this type in previous work [9]. Segmenting the network can reduce the vulnerability of the network to lateral movement of an insider threat or an adversary using stolen credentials. However, revoking the access of legitimate users will likely have an impact on productivity, especially if a user loses access to a computer they frequently use. For this reason, minimizing the total weight of the edges removed by the partition is desirable.

Table 8.3: Heuristic Search Parameters

Parameter	Value
Population size	60
Offspring size	60
Parent selection tournament size	2
Chance of mutation	25%
Chance of recombination	75%
Self-adaptive primitive level mutation chance	10%
Minimum initial parse tree depth	2
Maximum initial parse tree depth	5
Minimum mutation subtree depth	1
Maximum mutation subtree depth	3
Termination execution time limit	4 hours

Table 8.4: Data Set Summary

Authentication	
Unique users	9,924
Unique computers	14,822
Unique user-computer pairs	106,693
NetFlow	
Unique devices	60,185
Unique communication pairs	1,136,854

The second application models communication sessions between pairs of computers on the network. This information is collected as NetFlow sessions, which record the involved hosts as well as the length (elapsed time) and size (amount transferred) of the communication session. Edges in these graphs are weighted by the total size in megabytes of all sessions between pairs of computers. In this application, an edge removal would correspond to blocking communication between these pairs of computers.

Segmenting this style of network can improve security in a number of ways. Having separate disconnected domains can help compartmentalize the network and prevent unnecessary information leakage. A well segmented network is also less susceptible to the spread of malicious software. Graph partitioning methods have also been used to identify critical paths within the network for placement of intrusion detection monitors.

Of course, a graph partitioning approach is not the sole component of a security solution in either of these applications. Low-cost partitions can be used to identify the minimal effort plans for segmenting the network, but other factors will need to be considered when applying such a segmentation. For instance, when a user's ability to access a computer is revoked, a new computer might need to be allocated to replace the lost functionality. The factors involved with a complete network segmentation solution will vary by the application. This work focuses on the graph partitioning component of these network segmentation applications.

For both applications, a week's worth of data collected from the LANL network is used to construct a single graph [88]. A collection of subgraphs is created from each of these graphs using repeated randomized walks. This process ensures that the graphs used during evaluation of evolved partitioning heuristics are small enough to allow numerous evaluations while still maintaining characteristics of the original graphs. Previous work has shown that improvements in tailored graph partitioning heuristics can still be seen when scaling to graphs larger than those used during training [11].

The evolutionary searches are tasked with producing novel graph partitioning heuristics for each application. In this work, these take the form of bisection, or two-way partitioning algorithms. Partitionings with more than two partitions can be accomplished by repeated applications of a bisection algorithm. Evolved heuristics are compared against METIS, a popular general-purpose multi-level partitioning algorithm [19]. In order to demonstrate the impact of specialization, the heuristics evolved for each application are also applied to graphs produced from the opposite application. To measure the benefit of the self-adaptive dynamic primitive granularity control, three additional evolutionary searches are run using primitive operation sets that remain static throughout evolution. The **StaticLow** and **StaticHigh** configurations use the low-level and high-level primitive operation sets, respectively, while the **StaticCombined** configuration uses the combination of the two operation sets.

8.5 Results and Discussion

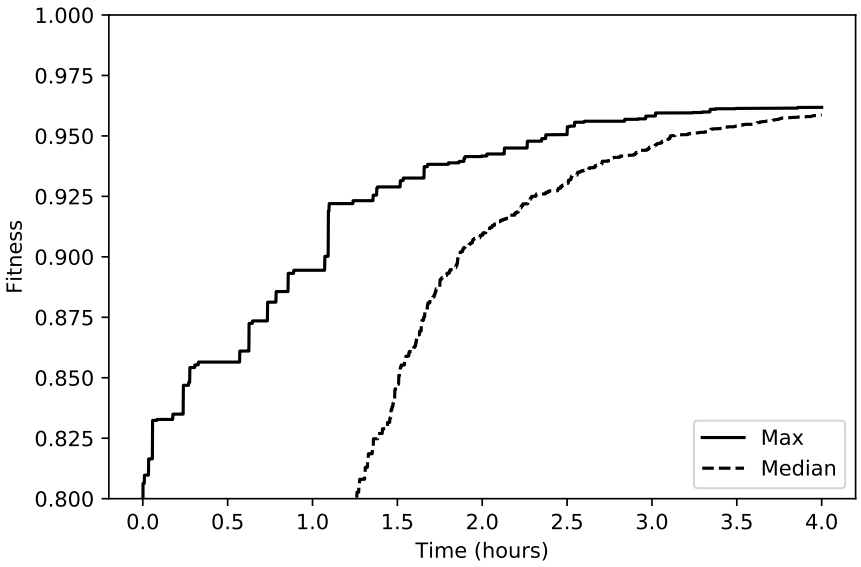
Figure 8.5 shows the increases in fitness during the evolutionary search for both applications using the **SelfAdaptive** configuration. The trend lines indicate the mean maximum and median

population fitness values over five runs of the GP. With access to high-level primitives that represent the functionality present in algorithms such as METIS, relatively high fitness values are seen very early in the run. For this reason, the range of the vertical axis is limited to highlight the gradual fitness increases seen later in the search. In both applications, the gains in the best fitness seen become very minor in the last hour of the search. However, the median population fitness values have clearly not converged. Future work will investigate the impact of longer evolutionary runs.

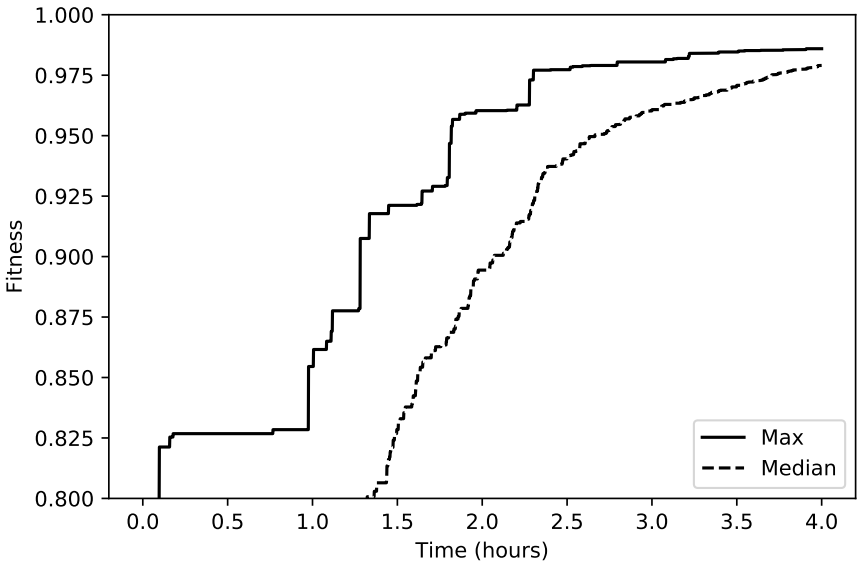
Figure 8.6 shows a comparison of the fitness values achieved for each method and application. In both cases, the heuristic evolved using the **SelfAdaptive** configuration to target the application consistently achieved higher fitness values than METIS, corresponding to lower cut-cost partitions. Also compared is the **OffTarget** solution evolved to target the opposite application. For these applications, it is clear that the performance improvement gained by tailoring the heuristics comes at the cost of generality. Not only does the **OffTarget** solution perform worse than the targeted solution, but it also does worse than the general-purpose METIS approach.

Although the static configurations did not perform as well as the **SelfAdaptive** approach, the **StaticHigh** configuration outperformed METIS for both applications, as did the **StaticCombined** configuration for the Authentication application. These configurations have access to the high-level graph partitioning operations inspired by other multi-level graph partitioning techniques, so this outcome is not unexpected. However, the fact that the **SelfAdaptive** configuration produced even better results suggests that the flexibility of dynamic primitive granularity control is beneficial. The fact that the **StaticLow** configuration performed poorly is also unsurprising since it requires that the GP rediscover useful combinations of the low-level primitives to achieve more complex functionality.

Table 8.5 summarizes the fitness comparison results numerically. The bolded value indicates the best performing partitioning method. For both applications, the **SelfAdaptive** approach produces statistically significant improvements in average fitness values when compared to METIS and the other GP configurations. The comparisons are performed using paired T-tests at the $\alpha = 0.05$ significance level.



(a) Authentication



(b) NetFlow

Figure 8.5: Population fitness values seen versus execution time for both applications using the **SelfAdaptive** configuration. Values are averaged over five repeated evolutionary searches.

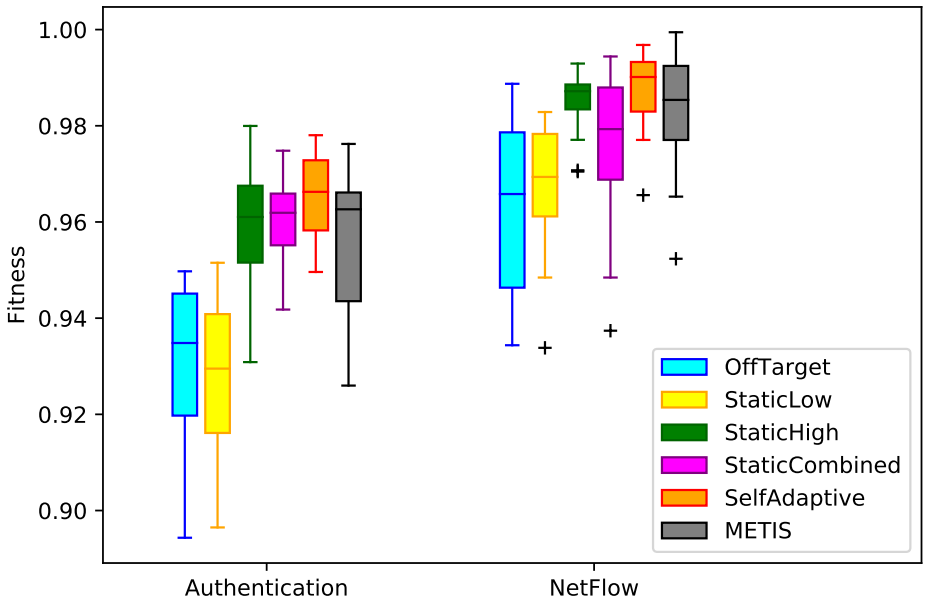


Figure 8.6: Comparison of average fitness values of partitions produced by each method for both applications. Costs are averaged over 30 applications on graphs unseen during evolution.

See Figure 8.7 for an example parse tree for a heuristic evolved to target the NetFlow application using the **SelfAdaptive** configuration. This example solution is set to the low primitive level, which explains why the parse tree only contains low-level primitives. However, closer inspection reveals several branches which are likely decomposed versions of higher level primitives. For instance, the right-most branch is simply the decomposed form of the KLRefinement primitive. The middle branch also resembles the high-level KLBisection primitive, but this heuristic has replaced the random ordering used to build the initial bisection with one based on node weight and degree. This level of fine tuning would not be possible without dynamic primitive granularity control unless the heuristic search discovered the high-level functionality on its own using low-level primitives.

8.6 Conclusion

Network segmentation is a tool that has applications in many domains, including computer and network security. Graph partitioning is often an invaluable component of network segmentation for identifying minimal-cost solutions. However, general-purpose off-the-shelf graph

Table 8.5: Partition Method Fitness Comparison

Authentication		
Method	**Mean**	**Variance**
OffTarget	0.93089	0.00025
StaticLow	0.92903	0.00021
StaticHigh	0.95844	0.00019
StaticCombined	0.96001	0.00008
SelfAdaptive	**0.96510**	0.00007
METIS	0.95539	0.00024
NetFlow		
Method	**Mean**	**Variance**
OffTarget	0.96292	0.00031
StaticLow	0.96839	0.00015
StaticHigh	0.98556	0.00003
StaticCombined	0.97700	0.00023
SelfAdaptive	**0.98787**	0.00005
METIS	0.98374	0.00012

partitioning methods do not leverage information about the specific characteristics of a network to find optimal partitioning solutions. Novel algorithms can be manually developed that improve performance by specializing for the types of graphs in question, but this process can be difficult and expensive. This work improves upon previous research that demonstrated the potential to automate the development of tailored graph partitioning heuristics by incorporating a dynamic approach to controlling the granularity of the heuristic search. Results from two real-world complex network applications demonstrate the potential of this approach to produce customized solutions that improve performance on the targeted application at the cost of generality. This technique has the potential to reduce the cost of network segmentation plans for application domains where frequent use of graph partitioning justifies the a priori search time needed to find an improved partitioning heuristic.

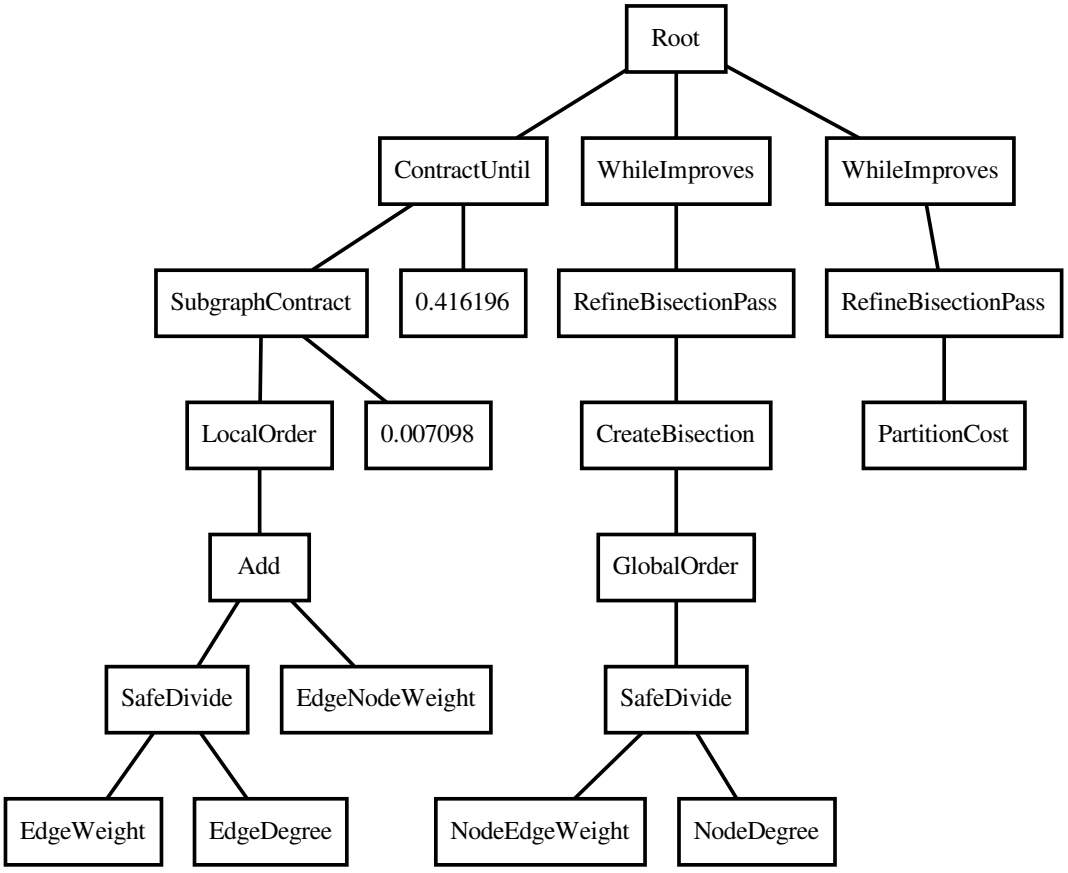


Figure 8.7: Example evolved graph partition heuristic parse tree for the NetFlow application.

Chapter 9

Conclusions

Due to their ability to naturally represent a variety of concepts, graphs are an invaluable tool in a number of application domains. As a result, many real-world problems can be tackled using graph-based approaches. This includes many cybersecurity challenges such as protecting computer networks against adversaries, identifying anomalous or malicious activity, or capturing the characteristics of benign activity in graph-based models. Previous approaches to addressing these problems often rely on general-purpose graph-based algorithmic solutions. These off-the-shelf methods exploit the ease of translating problems into a graph representation to find adequate solutions using existing techniques.

However, these graph algorithms are rarely designed with specific applications in mind and are often built for generality. Performance can be improved by leveraging additional problem-specific knowledge to tailor an algorithm to a certain application. This tailoring process can be done manually, but often requires significant domain expertise and development effort. The work presented in this dissertation demonstrates the potential of applying evolutionary and hyper-heuristic search techniques to automate the optimization of graph-based solutions to cybersecurity problems.

In chapters 2, 4, and 8, evolutionary methods are leveraged to improve performance on graph partitioning problems for network segmentation. Results demonstrate clear performance increases when exploiting problem-specific knowledge to tailor the graph partitioning process to the particular application of network segmentation. Although this customization is done manually in Chapter 2, chapters 4 and 8 show how performance gains can be achieved in an automated fashion through the use of hyper-heuristic searches.

Chapters 3 and 6 employ a similar hyper-heuristic approach to automate the design of graph models for various applications, including modeling computer network activity. In Chapter 5, a hyper-heuristic search is used to automatically generate network security metrics based on graph-based network attack models. Improving the accuracy of network modeling can be crucial when evaluating a network, testing graph-based networking algorithms, or making predictions concerning the structure or behavior of dynamic networks. These works show that accurate graph-based modeling and model analysis can be automated to reduce manual development time and reliance on subject matter expertise.

Finally, Chapter 7 demonstrates the use of a hyper-heuristic search to automate the design and optimization of novel link prediction algorithms for anomaly detection in computer and network security applications. Results show that predictive performance can be improved by automated algorithm customization to reduce false detection rates. This improvement in accuracy has the potential to dramatically reduce the effort expended investigating false positives and prevent the damage resulting from undetected malicious activity.

The work described in this dissertation involves contributions to the domains of evolutionary computation, hyper-heuristics, and cybersecurity. These works outline a general framework for automating the optimization of graph-based algorithms for improved performance on cybersecurity applications. The success of the hyper-heuristic techniques in this work relies on the careful cultivation of functionality from existing graph-based approaches. Chapters 3 and 7 demonstrate the performance impact of a well-constructed set of algorithmic building blocks.

Furthermore, Chapters 7 and 8 describe novel methods of dynamically controlling the level of heuristic search granularity. Results from these works demonstrate that these granularity control techniques have the potential to improve hyper-heuristic search performance on complex real-world applications. These improvements are unlikely to be restricted to graph algorithm or cybersecurity application domains. Dynamic primitive granularity control methods could contribute significantly to other complex automated algorithm design and optimization applications.

9.1 Research Impacts

Publications:

- "Evolving Random Graph Generators: A Case for Increased Algorithmic Primitive Granularity," in 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Dec. 2016 [10]
- "Evolving Multi-level Graph Partitioning Algorithms," in 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Dec. 2016 [11]
- "Automated Design of Network Security Metrics," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '18, (New York, NY, USA), pp. 1680–1687, ACM, 2018 [12]
- "Evolving Bipartite Authentication Graph Partitions," IEEE Transactions on Dependable and Secure Computing, vol. 16, no. 1, pp. 58–71, Jan/Feb 2019 [9]
- "Automated Design of Random Dynamic Graph Models," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, (New York, NY, USA), pp. 1504–1512, ACM, 2019 [13]
- "Automated Design of Tailored Link Prediction Heuristics for Applications in Enterprise Network Security," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, (New York, NY, USA), pp. 1634–1642, ACM, 2019 [14]
- "Automated Design of Tailored Link Prediction Heuristics using Dynamic Primitive Granularity Control for Applications in Enterprise Network Security" *in preparation for submission to IEEE Transactions on Dependable and Secure Computing*
- "Automated Design of Multi-Level Network Partitioning Heuristics Employing Self-Adaptive Primitive Granularity Control" *in preparation for submission to the 2020 Genetic and Evolutionary Computation Conference*

Contributions:

- Evolutionary approach to network access control that minimizes vulnerability with minimal impact on user productivity
- Hyper-heuristic framework for automating the design and optimization of novel graph-based algorithms for applications in computer and network security:
  - Generative graph models for complex networks, both static and dynamic
  - Tailored graph partitioning and network segmentation heuristics
  - Novel network security metrics for analyzing vulnerability to graph-based attack models
  - Customized link prediction heuristics for computer and network anomaly detection
- A novel hyper-heuristic approach for dynamically adjusting primitive operation granularity throughout a heuristic search
- Multiple methods of automatically controlling search granularity level to improve applicability and scalability for hyper-heuristic techniques for complex real-world problems

References

[1] A. Broido and K. C. Claffy, "Internet topology: connectivity of IP graphs," in Scalability and Traffic Control in IP Networks, vol. 45, pp. 172–187, International Society for Optics and Photonics, 2001.

[2] Y. Jin, E. Sharafuddin, and Z.-L. Zhang, "Unveiling Core Network-Wide Communication Patterns through Application Traffic Activity Graph Decomposition," ACM SIGMETRICS Performance Evaluation Review, vol. 37, no. 1, pp. 49–60, 2009.

[3] R. Perlman, "An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN," in ACM SIGCOMM Computer Communication Review, vol. 15, pp. 44–53, ACM, 1985.

[4] J. Dunagan, A. X. Zheng, and D. 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, SOSP '09, (New York, NY, USA), pp. 305–320, ACM, 2009.

[5] C. Phillips and L. P. Swiler, "A Graph-Based System for Network-Vulnerability Analysis," in Proceedings of the 1998 Workshop on New Security Paradigms, NSPW '98, (New York, NY, USA), pp. 71–79, ACM, 1998.

[6] P. D. Hough and P. J. Williams, "Modern Machine Learning for Automatic Optimization Algorithm Selection," in Proceedings of the INFORMS Artificial Intelligence and Data Mining Workshop, pp. 1–6, 2006.

[7] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," Journal of the Operational Research Society, vol. 64, no. 12, pp. 1695–1724, 2013.

[8] M. Illetskova, A. R. Bertels, J. M. Tuggle, A. Harter, S. N. Richter, D. R. Tauritz, S. Mulder, D. Bueno, M. Leger, and W. M. Siever, "Improving Performance of CDCL SAT Solvers by Automated Design of Variable Selection Heuristics," in Proceedings of the 2017 IEEE Symposium Series on Computational Intelligence (SSCI 2017), IEEE, Nov. 2017.

[9] A. Pope, D. Tauritz, and A. Kent, "Evolving Bipartite Authentication Graph Partitions," IEEE Transactions on Dependable and Secure Computing, vol. 16, no. 1, pp. 58–71, Jan/Feb 2019.

[10] A. S. Pope, D. R. Tauritz, and A. D. Kent, "Evolving Random Graph Generators: A Case for Increased Algorithmic Primitive Granularity," in 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Dec. 2016.

[11] A. S. Pope, D. R. Tauritz, and A. D. Kent, "Evolving Multi-level Graph Partitioning Algorithms," in 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Dec. 2016.

[12] A. S. Pope, R. Morning, D. R. Tauritz, and A. D. Kent, "Automated Design of Network Security Metrics," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '18, (New York, NY, USA), pp. 1680–1687, ACM, 2018.

[13] A. S. Pope, D. R. Tauritz, and C. Rawlings, "Automated Design of Random Dynamic Graph Models," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, (New York, NY, USA), pp. 1504–1512, ACM, 2019.

[14] A. S. Pope, D. R. Tauritz, and M. Turcotte, "Automated Design of Tailored Link Prediction Heuristics for Applications in Enterprise Network Security," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, (New York, NY, USA), pp. 1634–1642, ACM, 2019.

[15] B. C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks," IEEE Communications Magazine, vol. 32, pp. 33–38, Sept. 1994.

[16] C. Hummel, "Why Crack When You Can Pass the Hash," SANS Institute InfoSec Reading Room, vol. 21, 2009.

[17] "Microsoft Windows Kerberos 'Pass The Ticket' Replay Security Bypass Vulnerability." http://www.securityfocus.com/bid/42435. Accessed: 2016-08-10.

[18] A. E. Feldmann, "Fast Balanced Partitioning is Hard Even on Grids and Trees," in Proceedings of the 37th International Conference on Mathematical Foundations of Computer Science, MFCS'12, (Berlin, Heidelberg), pp. 372–382, Springer-Verlag, 2012.

[19] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," SIAM Journal on Scientific Computing, vol. 20, no. 1, pp. 359–392, 1998.

[20] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing, vol. 53. Springer, 2003.

[21] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197, 2002.

[22] A. D. Kent, L. M. Liebrock, and J. C. Neil, "Authentication graphs: Analyzing user behavior within an enterprise network," Computers & Security, vol. 48, pp. 150–166, 2015.

[23] A. Hagberg, A. Kent, N. Lemons, and J. Neil, "Credential Hopping in Authentication Graphs," in 2014 International Conference on Signal-Image Technology Internet-Based Systems (SITIS), IEEE Computer Society, Nov. 2014.

[24] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," Journal of Network and Computer Applications, vol. 36, no. 1, pp. 16–24, 2013.

[25] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-Attack Graph Generation Tool," in Proceedings of the DARPA Information Survivability Conference & Exposition II, 2001 (DISCEX '01), vol. 2, pp. 307–321, 2001.

[26] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, "Automated Generation and Analysis of Attack Graphs," in Proceedings, 2002 IEEE Symposium on Security and Privacy, pp. 273–284, 2002.

[27] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, Graph-based Network Vulnerability Analysis," in Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS '02, (New York, NY, USA), pp. 217–224, ACM, 2002.

[28] R. Ortalo, Y. Deswarte, and M. Kaâniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security," IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 633–650, 1999.

[29] W. Li and R. B. Vaughn, "Cluster Security Research Involving the Modeling of Network Exploitations Using Exploitation Graphs," in Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06., vol. 2, pp. 26–26, IEEE, 2006.

[30] R. Lippmann, K. Ingols, C. Scott, K. Piwowarski, K. Kratkiewicz, M. Artz, and R. Cunningham, "Validating and Restoring Defense in Depth Using Attack Graphs," in MILCOM 2006-2006 IEEE Military Communications Conference, pp. 1–10, IEEE, 2006.

[31] M. Uddin, A. A. Rahman, N. Uddin, J. Memon, R. A. Alsaqour, and S. Kazi, "Signature-based Multi-Layer Distributed Intrusion Detection System using Mobile Agents," International Journal of Network Security, vol. 15, no. 2, pp. 97–105, 2013.

[32] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in Proceedings of the 13th USENIX Conference on System Administration, LISA '99, (Berkeley, CA, USA), pp. 229–238, USENIX Association, 1999.

[33] W. Li, "Using Genetic Algorithm for Network Intrusion Detection," in In Proceedings of the United States Department of Energy Cyber Security Group 2004 Training Conference, pp. 24–27, 2004.

[34] I. Safro, P. Sanders, and C. Schulz, "Advanced Coarsening Schemes for Graph Partitioning," Journal of Experimental Algorithmics, vol. 19, no. 2, p. 24, 2015.

[35] C. Walshaw and M. Cross, "JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview," in Mesh Partitioning Techniques and Domain Decomposition Techniques (F. Magoules, ed.), pp. 27–58, Civil-Comp Ltd., 2007. (Invited chapter).

[36] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," Parallel Computing, vol. 34, no. 6, pp. 318–331, 2008. Parallel Matrix Algorithms and Applications.

[37] H. Meyerhenke, B. Monien, and T. Sauerwald, "A New Diffusion-based Multilevel Algorithm for Computing Graph Partitions of Very High Quality," in IEEE International Symposium on Parallel and Distributed Processing 2008, pp. 1–13, IEEE, 2008.

[38] J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon, "Genetic Approaches for Graph Partitioning: A Survey," in Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, pp. 473–480, ACM, 2011.

[39] M. Farshbaf and M.-R. Feizi-Derakhshi, "Multi-objective Optimization of Graph Partitioning using Genetic Algorithms," in Third International Conference on Advanced

Engineering Computing and Applications in Sciences, 2009. ADVCOMP'09., pp. 1–6, IEEE, 2009.

[40] P. Sanders and C. Schulz, "High Quality Graph Partitioning," Graph Partitioning and Graph Clustering, vol. 588, no. 1, 2012.

[41] A. J. Soper, C. Walshaw, and M. Cross, "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning," Journal of Global Optimization, vol. 29, no. 2, pp. 225–241, 2004.

[42] U. Benlic and J.-K. Hao, "A Multilevel Memetic Approach for Improving Graph k-Partitions," IEEE Transactions on Evolutionary Computation, vol. 15, pp. 624–642, Oct. 2011.

[43] T. Goel and N. Stander, "A Study on the Convergence of Multiobjective Evolutionary Algorithms," in Preprint submitted to the 13th AIAA/ISSMO conference on Multidisciplinary Analysis Optimization, pp. 1–18, 2010.

[44] A. D. Kent, "User-Computer Authentication Associations in Time." Los Alamos National Laboratory, 2014.

[45] J. D. Knowles and D. W. Corne, "Approximating the nondominated front using the Pareto Archived Evolution Strategy," Evolutionary Computation, vol. 8, no. 2, pp. 149–172, 2000.

[46] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," Physical Review E, vol. 64, p. 17, July 2001.

[47] P. Erdős and A. Rényi, "On random graphs I.," Publicationes Mathematicae, vol. 6, pp. 290–297, 1959.

[48] P. Erdös and A. Rényi, "On the Evolution of Random Graphs," Publ. Math. Inst. Hung. Acad. Sci, vol. 5, pp. 17–61, 1960.

[49] P. Erdös and A. Rényi, "On the Strength of Connectedness of a Random Graph," Acta Mathematica Hungarica, vol. 12, no. 1-2, pp. 261–267, 1961.

[50] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-world' Networks," Nature, vol. 393, no. 6684, pp. 440–442, 1998.

[51] M. Newman, Networks: An Introduction. New York, NY, USA: Oxford Univ. Press, 2010.

[52] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," Science, vol. 286, no. 5439, pp. 509–512, 1999.

[53] D. J. Price, "Networks of Scientific Papers," Science, vol. 149, no. 3683, pp. 510–515, 1965.

[54] S. Wasserman, Social network analysis: Methods and applications, vol. 8. Cambridge university press, 1994.

[55] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg, "Hyper-Heuristics: An Emerging Direction in Modern Search Technology," in Handbook of Metaheuristics (F. Glover and G. A. Kochenberger, eds.), vol. 57 of International Series in Operations Research & Management Science, pp. 457–474, Springer US, 2003.

[56] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," Natural Computing: an international journal, vol. 8, no. 2, pp. 239–287, 2009.

[57] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA, USA: MIT Press, 1992.

[58] A. Bailey, M. Ventresca, and B. Ombuki-Berman, "Genetic Programming for the Automatic Inference of Graph Models for Complex Networks," IEEE Transactions on Evolutionary Computation, vol. 18, no. 3, pp. 405–419, 2014.

[59] K. R. Harrison, "Network Similarity Measures and Automatic Construction of Graph Models using Genetic Programming," Master's thesis, Brock University, 2014.

[60] M. A. Martin and D. R. Tauritz, "Hyper-Heuristics: A Study On Increasing Primitive-Space," in Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion '15, (New York, NY, USA), pp. 1051–1058, ACM, 2015.

[61] D. J. Montana, "Strongly Typed Genetic Programming," Evol. Comput., vol. 3, no. 2, pp. 199–230, June 1995.

[62] S. Harris, T. Bueter, and D. R. Tauritz, "A Comparison of Genetic Programming Variants for Hyper-Heuristics," in Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1043–1050, ACM, 2015.

[63] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," Sociometry, vol. 40, no. 1, pp. 35–41, 1977.

[64] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[65] H. Meyerhenke, "Shape optimizing load balancing for MPI-parallel adaptive numerical simulations," Proceedings of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering, pp. 67–82, 2013.

[66] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, VLSI Physical Design: From Graph Partitioning to Timing Closure. Springer Science & Business Media, 2011.

[67] B. Peng, L. Zhang, and D. Zhang, "A survey of graph theoretical approaches to image segmentation," Pattern Recognition, vol. 46, no. 3, pp. 1020–1038, 2013.

[68] H. Li, G. W. Rosenwald, J. Jung, and C.-C. Liu, "Strategic Power Infrastructure Defense," Proceedings of the IEEE, vol. 93, no. 5, pp. 918–933, 2005.

[69] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-law Graphs," in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pp. 10–pp, IEEE, 2006.

[70] H. H. Hoos, "Automated Algorithm Configuration and Parameter Tuning," in Autonomous Search (Y. Hamadi, E. Monfroy, and F. Saubion, eds.), pp. 37–71, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[71] W. E. Donath and A. J. Hoffman, "Algorithms for Partitioning of Graphs and Computer Logic Based on Eigenvectors of Connections Matrices," IBM Technical Disclosure Bulletin, vol. 15, 1972.

[72] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell system technical journal, vol. 49, no. 2, pp. 291–307, 1970.

[73] S. B. Seidman, "Network Structure and Minimum Degree," Social Networks, vol. 5, no. 3, pp. 269–287, 1983.

[74] D. R. Tauritz and J. Woodward, "Hyper-heuristics Tutorial," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, (New York, NY, USA), pp. 510–544, ACM, 2017.

[75] P. Minarik and T. Dymacek, "NetFlow Data Visualization Based on Graphs," in Visualization for Computer Security (J. R. Goodall, G. Conti, and K.-L. Ma, eds.), (Berlin, Heidelberg), pp. 144–151, Springer Berlin Heidelberg, 2008.

[76] F. Harary and G. Gupta, "Dynamic Graph Models," Mathematical and Computer Modelling, vol. 25, no. 7, pp. 79–87, 1997.

[77] M. Turcotte, J. Moore, N. Heard, and A. McPhall, "Poisson Factorization for Peer-Based Anomaly Detection," in 2016 IEEE Conference on Intelligence and Security Informatics (ISI), pp. 208–210, Sept 2016.

[78] J. Neil, C. Hash, A. Brugh, M. Fisk, and C. B. Storlie, "Scan Statistics for the Online Detection of Locally Anomalous Subgraphs," Technometrics, vol. 55, no. 4, pp. 403–414, 2013.

[79] L. Boratto, S. Carta, A. Chessa, M. Agelli, and M. L. Clemente, "Group Recommendation with Automatic Identification of Users Communities," in 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, vol. 3, pp. 547–550, Sept. 2009.

[80] S. Eubank, H. Guclu, V. A. Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang, "Modelling Disease Outbreaks in Realistic Urban Social Networks," Nature, vol. 429, no. 6988, p. 180, 2004.

[81] X. Zhang, C. Moore, and M. E. J. Newman, "Random Graph Models for Dynamic Networks," The European Physical Journal B, vol. 90, no. 10, p. 200, Oct. 2017.

[82] P. Holme and J. Saramäki, "Temporal networks," Physics Reports, vol. 519, no. 3, pp. 97–125, 2012.

[83] K. R. Harrison, M. Ventresca, and B. M. Ombuki-Berman, "A meta-analysis of centrality measures for comparing and generating complex network models," Journal of Computational Science, vol. 17, pp. 205–215, 2016.

[84] M. R. Medland, K. R. Harrison, and B. M. Ombuki-Berman, "Automatic Inference of Graph Models for Directed Complex Networks using Genetic Programming," in 2016 IEEE Congress on Evolutionary Computation (CEC), pp. 2337–2344, July 2016.

[85] T. Menezes and C. Roth, "Symbolic Regression of Generative Network Models," Scientific reports, vol. 4, p. 6284, 2014.

[86] V. Arora and M. Ventresca, "Action-based Modeling of Complex Networks," Scientific Reports, vol. 7, no. 1, pp. 66–73, 2017.

[87] Y. Wang, A. Chakrabarti, D. Sivakoff, and S. Parthasarathy, "Fast Change Point Detection on Dynamic Social Networks," CoRR, vol. abs/1705.07325, pp. 1–8, 2017.

[88] M. J. M. Turcotte, A. D. Kent, and C. Hash, Unified Host and Network Data Set, ch. Chapter 1, pp. 1–22. World Scientific, nov 2018.

[89] Z. Wang, J. Liao, Q. Cao, H. Qi, and Z. Wang, "Friendbook: A Semantic-Based Friend Recommendation System for Social Networks," IEEE Transactions on Mobile Computing, vol. 14, no. 3, pp. 538–551, March 2015.

[90] G. Shani and A. Gunawardana, Evaluating Recommendation Systems, pp. 257–297. Boston, MA: Springer US, 2011.

[91] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," Proceedings of the National Academy of Sciences, vol. 99, no. 12, pp. 7821–7826, 2002.

[92] D. Liben-Nowell and J. Kleinberg, "The Link-Prediction Problem for Social Networks," Journal of the American Society for Information Science and Technology, vol. 58, no. 7, pp. 1019–1031, 2007.

[93] T. IDÉ and H. KASHIMA, "Eigenspace-based Anomaly Detection in Computer Systems," in Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, (New York, NY, USA), pp. 440–449, ACM, 2004.

[94] P. K. Gopalan, L. Charlin, and D. Blei, "Content-based recommendations with Poisson factorization," in Advances in Neural Information Processing Systems 27 (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3176–3184, Curran Associates, Inc., 2014.

[95] F. Caron and E. B. Fox, "Sparse graphs using exchangeable random measures," Journal of the Royal Statistical Society: Series B (Statistical Methodology), vol. 79, no. 5, pp. 1295–1366, 2017.

[96] L. Lü and T. Zhou, "Link Prediction in Complex Networks: A Survey," Physica A: Statistical Mechanics and its Applications, vol. 390, no. 6, pp. 1150 – 1170, 2011.

[97] F. Gonzalez, D. Dasgupta, and R. Kozma, "Combining Negative Selection and Classification Techniques for Anomaly Detection," in Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02, vol. 1, pp. 705–710, May 2002.

[98] G. P. Zhang, "Neural Networks for Classification: A Survey," IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 30, no. 4, pp. 451–462, Nov 2000.

[99] M. Meneganti, F. S. Saviello, and R. Tagliaferri, "Fuzzy Neural Networks for Classification and Detection of Anomalies," IEEE Transactions on Neural Networks, vol. 9, no. 5, pp. 848–861, Sept. 1998.

[100] A. Harter, A. S. Pope, D. R. Tauritz, and C. Rawlings, "Empirical Evidence of the Effectiveness of Primitive Granularity Control for Hyper-heuristics," in Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, (New York, NY, USA), pp. 1478–1486, ACM, 2019.

[101] P. Wang, B. Xu, Y. Wu, and X. Zhou, "Link prediction in social networks: the state-of-the-art," Science China Information Sciences, vol. 58, no. 1, pp. 1–38, Jan. 2015.

[102] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Temporal Link Prediction Using Matrix and Tensor Factorizations," ACM Trans. Knowl. Discov. Data, vol. 5, no. 2, pp. 10:1–10:27, Feb. 2011.

[103] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A Survey on Ensemble Learning for Data Stream Classification," ACM Computing Surveys, vol. 50, no. 2, pp. 23:1–23:36, Mar. 2017.

[104] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," Evolutionary Intelligence, vol. 1, no. 1, pp. 47–62, Mar. 2008.

[105] M. Suganuma, S. Shirakawa, and T. Nagao, "A Genetic Programming Approach to Designing Convolutional Neural Network Architectures," in Proceedings of the Genetic and

Evolutionary Computation Conference, GECCO '17, (New York, NY, USA), pp. 497–504, ACM, 2017.

[106] B. W. Goldman and D. R. Tauritz, "Meta-evolved Empirical Evidence of the Effectiveness of Dynamic Parameters," in Proceedings of the 13th annual conference companion on Genetic and evolutionary computation, pp. 155–156, ACM, 2011.

[107] J. R. Koza, D. Andre, F. H. Bennett III, and M. A. Keane, "Use of Automatically Defined Functions and Architecture-altering Operations in Automated Circuit Synthesis with Genetic Programming," in Proceedings of the First Annual Conference on Genetic Programming, pp. 132–140, Stanford University MIT Press, Cambridge, MA, 1996.

[108] P. J. Angeline and J. Pollack, "Evolutionary Module Acquisition," in Proceedings of the second annual conference on evolutionary programming, pp. 154–163, Citeseer, 1993.

[109] W. Banzhaf, D. Banscherus, and P. Dittrich, Hierarchical Genetic Programming Using Local Modules. Secretary of the SFB 531, 1999.

[110] J. P. Rosca and D. H. Ballard, "Hierarchical Self-organization in Genetic Programming," in Machine Learning Proceedings 1994, pp. 251–258, Elsevier, 1994.

[111] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," CoRR, vol. abs/1412.6980, 2015.