View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

DATA-CENTRIC METHODS FOR OPTIMIZATION AND PATTERN DISCOVERY IN NETWORKED SYSTEMS

BY

KARL HANY THOMPSON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Research Assistant Professor Huy Trong Tran

ABSTRACT

In this thesis, we examine two data-driven solutions to problems in operational networks. The first problem is concerned with assessing the resilience of the US air transportation network from an operational perspective. As a complex network comprising over 5,000 public airports and countless interfaces with other transportation systems, the impact of any disruption in the air network undoubtedly extends to other inter-connected economic and functional domains. Our solution to the resilience assessment problem is a tri-level optimization program that is able to simulate worst-case disruptions in the air network as well as propose the optimal ways to mitigate their effects. These mitigation steps take the form of investment recommendations for the air routes that are in most need of augmentation by other high-speed transportation modes. Our methodology and results for this application are explained in detail in Chapter 3. The second problem discussed in this thesis is centered on identifying design patterns in architecture graph representations of operational systems. Design patterns have been well documented and researched in software systems as a valuable design tool since the nineties. However, their usage has not been significantly expanded beyond software architectures, and their discovery methods have generally remained structured and supervised. We propose an end-to-end, unsupervised graph generation and pattern identification framework that is able to find unknown and potentially useful patterns in architecture graphs using machine learning. Our method is not limited to software systems, and is designed to be able to make possible pattern predictions even with a single architecture graph input. We detail our framework and experimental results in Chapter 4. Organizationally, Chapter 1 of the thesis starts with an introduction to network theory and graph representations, Chapter 2 provides background on network optimization and graph machine learning, and Chapter 5 concludes the thesis with our final thoughts and future research directions.

to my family, friends, and mentors.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: LITERATURE REVIEW 4

CHAPTER 3: RESILIENCE ANALYSIS

OF THE US AIR TRANSPORTATION NETWORK..25

CHAPTER 4: UNSUPERVISED PATTERN IDENTIFICATION

IN ARCHITECTURE GRAPHS 41

CHAPTER 5: CONCLUSIONS 60

REFERENCES 61

CHAPTER 1: INTRODUCTION

A network is defined as a group or system of interconnected people or things [1]. The world around us can be viewed as a network of networks. Everything from electrical and communication systems to social and biological groups represent networks that are both intra- and inter-connected to each other. Networks vary considerably by size and complexity, but they all share the same property that they are all composed of components of varying types, as well as connections between them. This thesis addresses two challenging problems related to the modeling and analysis of networked systems. The first relates to how the resilience of an infrastructure network can be assessed and improved. The second seeks to identify design patterns in network representations.

1.1 NETWORKS AS GRAPHS

In Network Theory, the mathematical representation of a network is a data structure called a graph. Graphs describe network components and the relationships between them. These relationships can be directed, as is the case with email communication that takes place between distinct senders and recipients, and they can be undirected, as in chemical molecules and biological networks. Mathematically, the relationships between graph components are modeled as edges (links), while the components themselves are modeled as nodes (vertices). For example, a twitter social circle is a type of network, and therefore it can be modeled as a directed graph where users are considered nodes and their connections considered edges. Using a subset of publicly available and anonymized data collected by McAuley and Leskovec [2], Figure 1 shows how a sample of fifty Twitter users are connected in their social circle in directed graph form. Graph nodes and edges can have different attributes, such as labels, that distinguish them apart.

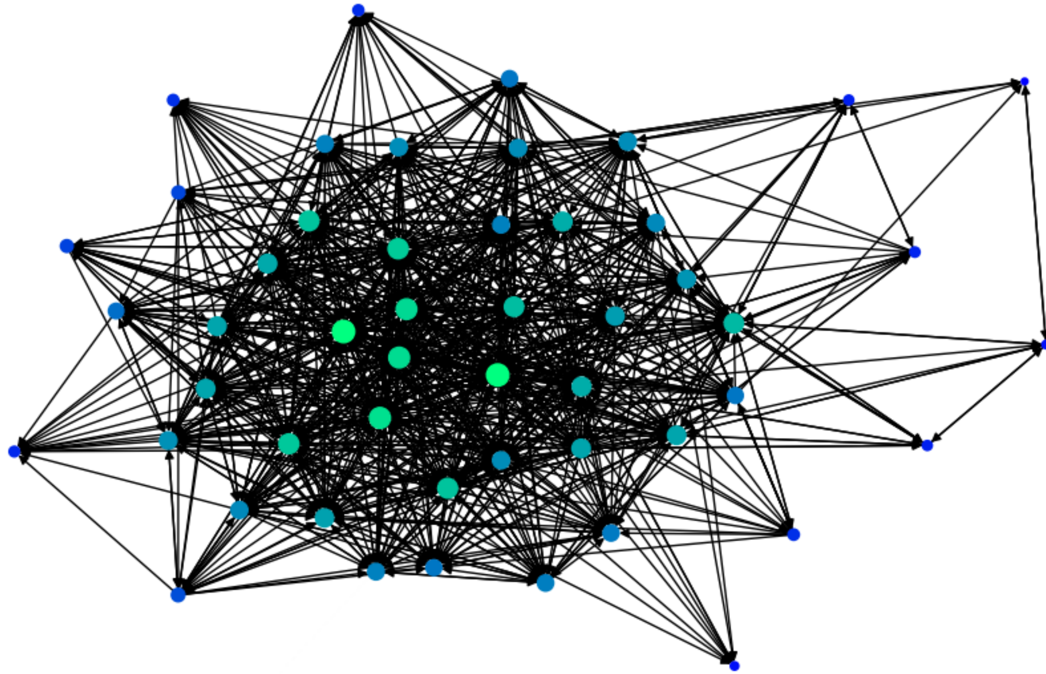

Figure 1: A sample of fifty anonymous Twitter users and their inter-connections. A node's size and color correspond to its number of connections.

Representing networks as graphs comes with a number of advantages. Graphs provide a standard way of modeling otherwise starkly different types of networks, which opens the way for conducting analyses of their inter-dependencies and design patterns in ways that would be fairly difficult otherwise. Additionally, graphs could be represented mathematically in multiple, inter-changeable forms, which makes them compatible with complex applications that comprise multiple algorithms with varying input specifications. Of these mathematical forms, edge lists are the simplest way to represent a graph; they describe all edges in the graph sequentially, identifying each edge by the two nodes that it is incident on. Adjacency matrices are another common form; they are two-dimensional arrays in which each row and column correspond to a node, and the value of each array element is either one or zero for unweighted graphs, depending on whether or not there is a connection between the row and column nodes. Another form is adjacency lists, in

which each row corresponds to a graph node and contains the identities of the other nodes that are connected to it. These compact representations enable the application of rigorous mathematical methods to analyze and model graphs, from their structural characteristics to behavioral ones. Combined, these reasons position graph data structures as a uniquely suitable mathematical form for the study of networked systems.

In this thesis, we examine two distinct applications that benefit greatly from the representation of networks in graph format. The first application, presented in Chapter 3, centers on analyzing the resilience of the US air transportation system from a network optimization perspective. In this framework, the air network is modeled as a large undirected graph with airports modeled as nodes and air routes as edges. The main goal of the optimization problem is to identify the specific edges that are most critical to the overall resilience of the network. The second application, detailed in Chapter 4, attempts to discover design patterns in operational system architectures in an unsupervised manner using machine learning. This is facilitated through the representation of the architectures as directed graphs, with system components modeled as nodes and their connections as edges. By breaking down a given architecture graph into its fundamental subgraph set, we are able through clustering techniques to identify the subgraphs that have the highest potential of comprising design patterns. We finally note our conclusions and final thoughts in Chapter 5.

CHAPTER 2: LITERATURE REVIEW

Network science, as an academic field that includes such disciplines as network theory and complex network analysis, has been an increasingly important component in modern engineering practice. Origins of network science date back to the eighteenth century, with one of the earliest well-known problems in the field, the Seven Bridges of Königsberg, being formulated by Leonhard Euler in 1736 [3]. Since then, the field has grown exponentially, alongside related sciences such as mathematics, statistics, physics, and computer science to be the complex field that it is today. Network science is commonly used at present to study transportation, communication, computational, biological networks, and others. Methods used to study these networks include optimization, link analysis, social simulation, multi-agent systems, and dynamic network analysis. This chapter examines a select group of network optimization and graph machine learning methods in preparation for our air network resilience and architecture graph pattern problems in Chapters 3 and 4, respectively.

2.1 NETWORK OPTIMIZATION

Is his book "Network Optimization: Continuous and Discrete Models," [4] Dimitri Bertsekas discusses network optimization as an intersection of linear programming and combinatorial optimization. He approaches the problem of studying network optimization models from two angles: the linearity of the model (linear or nonlinear), and whether it is discrete or continuous. The book discusses several optimization theory problems that fall within the scope of these angles. One such problem often formulated as a linear model is the *shortest path problem*, where the objective is to find a path between two vertices in a given graph in a way that minimizes

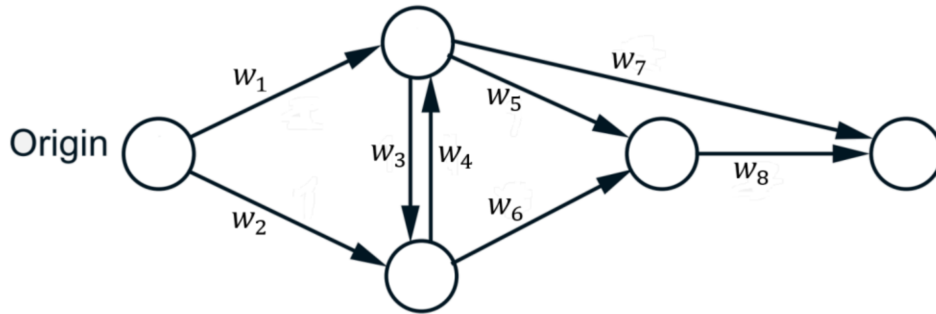

Figure 2: A standard shortest path problem where the objective is to find the smallest distance from the origin node to all other nodes.

the collective cost incurred by traversing edges with set weights. Practical applications involving this problem include: the routing of information in data networks from computer nodes through communication links, the completion of work tasks in a specific order to achieve project goals optimally, and the breaking down of a given word paragraph into a number of successive lines to improve readability. The most notable early algorithms that were developed to solve this problem are Dijkstra's algorithm [5], and the Bellman–Ford algorithm [6]. Dijkstra's algorithm runs in time proportional to the square of the number of nodes and only considers non-negative edge weights in finding the shortest path between a single source node and all other nodes in the graph. The Bellman–Ford algorithm, on the other hand, has a longer run time that is proportional to the number of graph nodes times the number of edges, but improves over Dijkstra's algorithm with its ability to generalize to graph edges with negative weights in its calculation of the shortest path. An abstract example of a simple shortest path problem is shown in Figure 2.

Another well-known linear problem discussed in the book is the *max-flow problem*, in which the objective is to transport as much flow as possible in a graph from an origin node known as the *source* to a destination node known as the *sink* while staying within each edge’s capacity constraint. The problem appears in practical applications such as the calculation of network

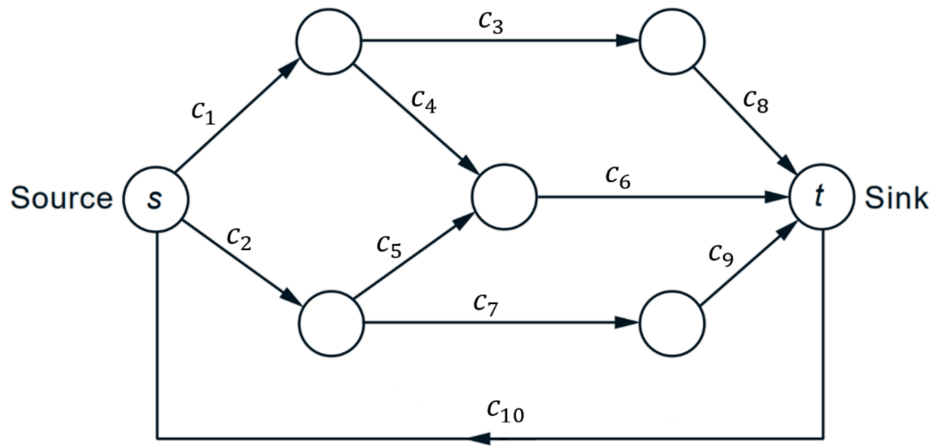

Figure 3: An example of a max-flow problem with ten edges where the objective is to move the highest possible flow from the source node to the sink node.

throughput for transportation and communication systems, and the scheduling of flight crews in the airline industry. There exist many algorithms to solve the problem, among them are the Ford–Fulkerson algorithm [7] and Dinic's algorithm [8]. The Ford–Fulkerson algorithm is a greedy algorithm that works by successively moving flow over augmenting paths, that is paths that have available capacities on all their edges, until no such paths exist. The time complexity of the algorithm is on the order of magnitude of the maximum flow times the number of edges, which for large networks can be quite prohibitive. On the other hand, Dinic's algorithm works by finding all shortest augmenting paths in one step, and the resulting running time is on the order of magnitude of the number of nodes squared times the number of edges. Figure 3 shows a simple max-flow application problem.

Closely related to the max-flow problem is the *min-cost flow problem*. While the former focuses on sending the maximum possible amount of flow through the network, the latter focuses on finding the cheapest way to send a specific amount of flow through the network instead. A direct application of this problem is the task of finding the optimal route to transport goods between

factories and warehouses in an industrial network where the roads have some capacity and cost constraints associated with them. In relation to the two previously discussed problems, the min-cost flow problem can be viewed as a derivative of both; we can reduce it to the shortest path problem if we remove the edge capacity constraint, and we can reduce it to the max-flow problem if we set the edge weights to zero. However, unlike previously discussed problems, the min-cost flow problem can be solved fairly efficiently in polynomial time using the network simplex algorithm [9].

All the problems examined so far have been *continuous* in that their solutions were allowed to take fractional values. Problems that require solutions to have integer values are called *integer programming* problems. Such problems are commonly used to model optimization problems that aim to find optimal discrete decisions, such as the optimal hourly production rate in a factory, or the optimal locations to position warehouses within a distribution network. We examine next the basics of integer programming and some recent applications.

2.1.1 Integer optimization

The idea of integer programming is to minimize or maximize an objective function where all or some of the variables are limited to be integers. The problems where all variables are integers are called *pure integer programs*, and those where only some variables have to be integers are called *mixed-integer programs*. A simple linear integer optimization problem could take the form:

$$Maximize \sum_{j=1}^{n} c_j \, x_j$$

subject to:

$$\sum_{j=1}^{n} a_{ij} \, x_j = b_i, \qquad i = 1, 2, \dots, m$$

$$x_j \geq 0, \qquad j = 1, 2, \dots, n$$

$$x_j \ integer. \qquad j = 1, 2, \dots, n$$

where the objective is to maximize the sum of the quantity $c_j \, x_j$ for all $j \leq n$. In this case, the values $a, b, and \ c$ are constants, and x is the only decision variable which we're interested in calculating. The constraints mandate that all x values be positive and integer, and that the sum of $a_{ij} \, x_j$ for all $j \leq n$ be equal to b_i for $i \leq m$.

In literature, linear integer programming has been used to study problems in a variety of industries. Zhu et al. [10] develop an integer linear programming-based optimization model to study residential electrical load management in a smart grid. Their proposed consumption scheduling framework seeks to minimize the highest hourly electrical load in order to reach a balanced daily load scheme. The algorithm is formulated in the form of a linear integer program as:

$$\min_{L, x_{a,h} \in \mathbb{R}} L$$

subject to:

$$\sum_{a \in A} x_{a,h} \leq L, \qquad \forall h \in H,$$

$$1^t \, x_a = l_a, \qquad \forall a \in A,$$

$$x_{a,h} \geq 0.$$

where L is the peak hourly load, x is the schedule plan, a is an appliance out of the set of all appliances A, h is the hour of day out of the set of all hours H, and l is the daily power requirement for each appliance. In their numerical simulations, the algorithm, guided by the appliances' power consumption history, succeeds in calculating the optimal operating power and time for electrical appliances connected to the smart grid.

In a different direction, Richards et al. [11] use a mixed-integer linear program to develop an aircraft collision avoidance and trajectory planning algorithm. The collision avoidance task of the algorithm is achieved using a reduced aircraft dynamics model with purely linear constraints to simulate a simplified version of an aircraft's flight pattern. The trajectory planning task for a multi-aircraft simulation is simply accomplished by minimizing the sum of travel times for each vehicle. Its mixed-integer linear program implementation in its simplest form is given as:

$$minimize\ J = \sum_{i=1}^{T} \sum_{p=1}^{N} T_i\, b_{ip}$$

subject to:

$$x_{ip} - x_{F_p} \leq R\left(1 - b_{ip}\right), \qquad \forall\, p \in [1, \dots, N], i \in [1, \dots, T]$$

$$x_{ip} - x_{F_p} \geq -R\left(1 - b_{ip}\right), \qquad \forall\, p \in [1, \dots, N], i \in [1, \dots, T]$$

$$y_{ip} - y_{F_p} \leq R\left(1 - b_{ip}\right), \qquad \forall\, p \in [1, \dots, N], i \in [1, \dots, T]$$

$$y_{ip} - y_{F_p} \geq -R\left(1 - b_{ip}\right), \qquad \forall\, p \in [1, \dots, N], i \in [1, \dots, T]$$

$$\sum_{i=1}^{T} b_{ip} = 1. \qquad \forall\, p \in [1, \dots, N]$$

where T_i is the time elapsed at time step i, b_{ip} is a binary variable that only takes a value of 1 of vehicle p reaches its destination in time step i, (x_{ip}, y_{ip}) is the position of aircraft p at time step i, (x_{F_p}, y_{F_p}) is the position of the destination, and R is an arbitrary large positive number. Effectively, the program requires each aircraft p to reach its destination at some time step i that is earlier than the maximum allowable time T. In their experiments, the authors demonstrate their algorithms on three experiments involving a single aircraft with fixed start and end points, multiple aircraft with fixed start and end points, and a single aircraft with multiple waypoints. The results show that the computed flight trajectories are consistent with real flight dynamics and physical limitations.

In the electrical power production field, Carrion et al. [12] develop a mixed-integer linear optimization program to efficiently solve the thermal unit commitment problem. Such a problem involves computing a schedule of when to start and turn off thermal generating units, and determining when to activate online generators temporarily in order to satisfy system demand and minimize operational cost while abiding by set power generation constraints. Their mathematical formulation takes the form:

$$minimize \sum_{k \in K} \sum_{j \in J} c_j^p(k) + c_j^u(k) + c_j^d(k)$$

subject to:

$$\sum_{j \in J} p_j(k) = D(k), \qquad \forall\, k \in K$$

$$\sum_{j \in J} \overline{p_j}(k) \geq D(k) + R(k), \qquad \forall\, k \in K$$

$$p_j(k) \in \Pi_j(k), \qquad \forall\, j \in J, k \in K$$

where $c_j^p(k)$, $c_j^u(k)$, $c_j^d(k)$ are the production cost, startup cost, and shutdown cost of thermal unit j in time period k, respectively. Combined, the three variables represent the overall operation cost that the algorithm seeks to minimize. The variables $p_j(k)$ and $\overline{p_j}(k)$ are the power output and maximum available power output, respectively, for thermal unit j in time period k. The constants $D(k)$ and $R(k)$ represent the load demand and operating energy reserve requirement, respectively. Traditionally, Lagrangian relaxation, in which a simpler problem is derived and solved from the original problem and taken as an approximation, was commonly used to solve the unit commitment problem due to its ability to scale to large problems. Its main disadvantage, however, lies in its dependence on heuristics to discover feasible solutions. The main advantage of applying integer linear programming to this problem is that due to its NP-complete nature, the program is

guaranteed to reach the optimal solutions in polynomial time. The authors confirm this improvement over traditional methods in their numerical results.

Moving beyond the linear problems mentioned so far, many practical optimization tasks involve complex models in which the objective function and constraints may be nonlinear. Such problems appear in a wide range of disciplines such as control systems design, signal processing, structural optimization, operations research, and others. In the next section, we examine some of these models and applications.

2.1.2 Nonlinear integer programming

Nonlinear programming differs than linear programming in that while linear programming problems require the existence of constraints to reach a solution, nonlinear problems can either be constrained or unconstrained [13]. In a general case, a mixed-integer nonlinear optimization model could take the form [14]:

$$maximize\ f(x_1, \ldots, x_n)$$

subject to:

$$g_1(x_1, \ldots, x_n) \leq 0,$$

$$\vdots$$

$$g_m(x_1, \ldots, x_n) \leq 0,$$

$$\mathbf{x} \in \mathbb{R}^{n_1} \times \mathbb{Z}^{n_2}$$

where the functions f, $g_1, \ldots, g_m$ are arbitrary nonlinear functions. Unlike linear programs, the optimal solution to this problem is not limited to being one of a finite number of feasible points. It follows that nonlinear programs are much harder to solve, and often require algorithms specialized to the types of problems they model. Established algorithms that have been used to solve nonlinear

integer programs include generalized Benders decomposition [15], branch and bound [16], and outer-approximation [17].

Among many disciplines, nonlinear programming has been historically successful in tackling complex problems in network design. Bragalli et al. [18] apply a mixed-integer nonlinear optimization program to the problem of designing a water distribution network. The goal of their model is to select the optimal pipe diameters for all the pipes that comprise the distribution network in a way that minimizes the cost of constructing the network, while accounting for hydraulic constraints. The model's objective function is of the form:

$$minimize \sum_{e \in E} C_e \left(D(e)\right) len(e)$$

where C_e is a continuous cost function, $D(e)$ is a discrete variable representing the diameter of pipe e, and E is the set of all pipes. The model's physical constraints are formulated as:

$$d_{min}(e) \leq D(e) \leq d_{max}, \qquad \forall\, e \in E$$

$$p_{min}(i) + elev(i) \leq H(i) \leq p_{max}(i) + elev(i), \qquad \forall\, i \in N$$

the first of which limits a pipe's diameter to a certain permissible range, while the second limits the hydraulic head of junction i to a range determined by the minimum and maximum gauge pressure and the pressure due to elevation. The model also contains flow constraints that set limits for flow rates and head loss across links, and ones that ensure flow conservation. Computationally, the model is solved through a relaxation of the nonlinear model and a reparameterization of pipes by cross-sectional area. Results show that the developed model produces better computational estimates for the water distribution network problem than linear models, in part due to its better representation of nonlinear flow mechanics that are nonlinear in nature.

In a similar application, Martin et al. [19] develop a mixed-integer nonlinear program to optimize the flow in a gas network. Similar to the min-cost flow problem, the goal of this model

is to deliver gas to consumers in a large network that contains variable sources and demands. The nonlinear complications of the problem arise from the physical friction between the gas and network pipes, which results in pressure loss and requires the use of gas-powered compressors. Hence, the objective of the nonlinear model is to satisfy gas demand across the network in a way that minimizes compressor gas consumption. The model takes into account gas flow mechanics, pipe characteristics, valve flow bounds, and compressor fuel consumption. To be viable computationally, the developed model is simplified using piece-wise linear approximation that reduces it to a sequence of linear functions by adding extra variables and constraints to the model formulation. Results show that the gas flow metrics and compressor usage obtained from the reduced model are comparable to practical values for real-sized networks.

In an electrical engineering context, Murray et al. [20] attempt to approach the problem of determining the sizes and locations of electrical substations on a rectangular electrical grid from an integer programming perspective. Their developed model has a nonlinear objective function with linear constraints, and is solved using a series of local relaxations. The full formulation is given as:

$$\min_{V,\bar{I},y} C_{cap}\, e^T\, y + C_{loss}\, V^T\, Y\, V$$

subject to:

$$\bar{I} - YV = 0,$$

$$V^l \leq V_i \leq 1,$$

$$\bar{I_\iota} \leq S_{cap}, y_i = 1,$$

$$\bar{I_\iota} = l_i, y_i = 0,$$

where C_{cap} is the cost of a new substation, $e^T\, y$ is the number of new substations, C_{loss} is the network losses, $\bar{I}, V$ and l are vectors of currents, voltages and loads at network nodes, respectively,

Y is the admittance matrix for network linkages, and S_{cap} is the capacity for new substations. The objective is to minimize the sum of the cost of building new substations and the cost of electrical losses. Computational results show that through local relaxation the developed algorithm performs comparably to commercial solvers such as SBB and CPLEX in some cases at significantly shorter run times.

All the models and applications discussed so far are generally based on mathematical optimization and can be applied to a wide variety of problems. Chapter Three's topic, analyzing the resilience of the U.S. air transportation networks against intelligent attacks from an optimization standpoint, requires some additional background on its mathematical foundation and the problem it models. We present this background and relevant papers from the literature next.

2.1.3 Optimization for air network resilience

Applying optimization models to the analysis of various infrastructure networks has been an ongoing effort in the field of operations research for many years. The U.S. Government's interest in identifying and protecting critical infrastructure systems was codified in President Clinton's Commission on Critical Infrastructure Protection Report [21], in which it defined critical infrastructures as: "Infrastructures so vital that their incapacitation or destruction would have a debilitating impact on defense or economic security." Since then, the government has allocated significant resources towards analyzing and providing policy recommendations for the protection of these systems.

Among the various approaches to analyzing these complex networks, Brown et al. [22] introduce a novel multi-objective optimization model named *defender-attacker-defender* that is formulated as a three-stage Stackelberg game [23]. In the model, three competing agents seek to

optimize their respective objective functions as independent decision makers. Applied to a general network with commodity movement, the first defender agent's task is to examine supply and demand at each node and designs transportation plans in a way that minimizes operational cost. The attacker agent's task is to try to increase that cost by disrupting key network links. In turn, the second defender agent, which acts as a protector, attempts to minimize the damage of the potential attack by protecting existing network links or establishing new ones. Mathematically, the model is formulated as:

$$\min_{w \in W} \max_{x \in X} \min_{y \in Y} cy$$

subject to:

$$Ay = b,$$

$$0 \leq y \leq \mathcal{U}(1 - (x - w)^{+}),$$

where w, x, and y are the defense, attack and routing plans, respectively chosen from the sets of all available plans W, X, and Y. The constant c is a vector representing the network's operating costs and penalties. In the first constraint equation, A is the network's adjacency matrix, and the variable b contains the supply and demand at each network node. The second constraint equation introduces $\mathcal{U}$, a matrix representing network flow capacity limitations, to place an upper limit on the maximum allowable flow routing under given defense and attack plans.

Brown et al.'s formulation, thanks in part to its minimal and non-subject-matter-specific constraints, opened the door for various bi- and tri-level optimization programs representing competing agents to be applied in the context of infrastructure systems analysis. Among these studies, Salmeron et al. [24] develop a generalized Benders decomposition scheme to solve a bi-level network interdiction problem that attempts to model the non-concave power routing requirements in a large-scale electric grid. The goal of the program is to identify the set of network

components whose loss will have the highest negative impact on residential customers in terms of economic losses. The model also includes a subproblem that explores the optimal power flow routing that can be utilized at various stages of network repair after the loss. To test their formulation, the authors use a regional US power grid with 500 generators, 6000, lines and 5000 buses, and show with numerical results that their algorithm is able to identify the most critical components in the network that would cause the highest long-term impact.

In a different direction, Pita et al. [25] develop a multi-agent algorithm that casts the Los Angeles International Airport security system as a Bayesian Stackelberg game and recommends randomized patrolling patterns, taking into consideration their complex costs and benefits. Their mixed-integer quadratic programming formulation uses an algorithm called Decomposed Optimal Bayesian Stackelberg Solver (DOBSS) that is able to achieve an optimal, non-approximate solution for the problem by accounting for multiple adversary models. Their algorithm was implemented successfully at the Los Angeles International Airport in 2007 to establish optimal random locations for checkpoints on airport roadways as well as canine patrol routes.

Boardman et al. [26] formulate a two-player, three-stage sequential program that optimizes the positioning of a surface-to-air missile defense battery system in response to a hypothetical missile attack launched by an intelligent adversary. The positioning algorithm takes into account the attacker's future knowledge of battery location and possible subsequent attack plans based on this information. Additionally, the last stage of the model represents a defender program that is tasked with calculating the optimal assignment of interception missiles from positioned batteries in response to an attack. The model variables are the number of cities defended, the number of missile battery types, the number of interceptor missile types, and the total number of attacker missiles. Since full enumeration is not practical for such large problems, the authors use a tree

search technique called Double Oracle that is able to compute an exact Nash equilibrium for two-player sequential games. The authors test their model on 52 instances with varying parameters and find that the obtained solution is consistently within 3% of the optimal solution for all their experiments.

Alderson et al. [27] examine the resilience of the San Francisco Metro Area highway network through a tri-level defender-attacker-defender program that accounts for nonlinear congestion behavior and realistic travel demands. The model is designed to analyze scenarios where a loss of one or more road segment, bridge, and/or tunnel may occur in a way that impedes traffic. In such cases, travelers, represented in the model as an optimization agent, are able to change their routes dynamically to avoid the congestion, or cancel their travel plans altogether. The goal of the algorithm is to identify the highway network segments that would result in the longest travel time increase when impeded. The authors conclude that the location of the lost highway segment within the overall network and the restoration time needed to restore its operability after a loss are key to evaluating a segment's criticality to network resilience.

The literature explored thus far serves as a necessary background for the air network resilience problem discussed in Chapter 3. Continuing with the literature review, we examine next some machine learning methods and their applications on graph-structured networks that will serve as a basis for the discussion in Chapter 4.

2.2 NETWORK MACHINE LEARNING

Machine learning in its simplest form refers to the ability of programmed computers to detect meaningful patterns in datasets [28]. Machine learning has shown significant success in performing tasks that are relatively easy for humans to do, but very hard for traditional

programming methods and algorithms. Examples of these tasks include image classification [29], control of autonomous vehicles [30], and even playing video games [31]. Many of these advancements are made possible through the use of artificial neural networks, which form the basis for a subset of machine learning called *deep learning*.

Deep learning has been used in recent years to analyze and process graph and network data for different applications. Graph deep learning methods can be classified into three types: supervised, semi-supervised, and unsupervised. The latter two types are often less mature than supervised methods, but are generally more applicable to network data that often do not have clear labels. Semi-supervised learning is a subset of deep learning that utilizes both labeled and unlabeled data for training. Unlike supervised learning, which requires the full training dataset to be labeled, semi-supervised methods are able to generalize from the usually small amount of labeled data and apply the learned data characteristics to the unlabeled data. In contrast, unsupervised learning attempts to draw inferences from the training data without relying on any labeling scheme.

2.2.1 Semi-supervised graph learning

Current semi-supervised learning methods on graphs can be divided into three categories: graph neural networks, graph convolutional networks, and graph reinforcement learning [32]. The goal of graph neural networks is to encode the structural information of networks in graph format such that the adjacency of each node in the graph is reduced to a low-dimensional state vector [33]. Battaglia et al. [34] use the graph neural networks framework to construct a reasoning model of how objects in dynamic systems interact and influence each other. In their interaction network model, objects are represented as nodes and their relations are represented as edges in a graph that

aims to encompass the properties of a complex system. The purpose of the study is to emulate physical collision and rigid and non-rigid body mechanics through deep neural networks instead of rule-based physical simulations. The experimental results show that the developed model is able to accurately predict the dynamic trajectories of physical objects, estimate their potential energy, and generalize to systems with different objects and relations.

Graph convolutional networks (GCNs) are another family of semi-supervised learning methods that has been developed by generalizing convolutional neural networks to operate on graph-structured data instead of images and grids. The idea of convolution networks is to learn a representation of the input data, in the form of weights and biases, through a series of mathematical operations. Graph convolutional networks, in particular, aim to learn a function to generate a representation of every node in an input graph while taking into account its own as well as its neighbors' features. Kipf et al. [35] use a GCN to classify node types of several citation network datasets, and show that their model results outperform other classification algorithms such as DeepWalk [36] and SemiEmb [37].

A typical graph convolutional network is composed of several layers that the graph information flows through, from the input layer that accepts graph adjacency and node feature matrices to the output layer that contains the predicted node labels. In [35], the flow of information from layer to layer in the GCN is governed by the rule:

$$H^{(l+1)} = \sigma\left(\widetilde{D} - \frac{1}{2}\tilde{A}\widetilde{D} - \frac{1}{2}H^{(l)}W^{(l)}\right)$$

where $\tilde{A}$ is the sum of the graph adjacency matrix A and the identity matrix. The array $\widetilde{D}$ is the degree matrix for the graph, where for each node i, the degree component is calculated by $\widetilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, where j iterates over all other network nodes. The array $H^{(l)}$ represents the activations of layer l. For the network's input layer, the activation matrix $H^{(0)}$ is equal to the node feature matrix

X. The array $W^{(l)}$ is the trainable weight matrix of layer l. Together, the weight matrices of all network layers form the main mechanism that enables the network to learn graph distributions. As a final step, the calculations in the parenthesis gets passed to an activation function σ that performs a final mathematical operation on the activation values before passing the signal to the next network layer. The specific mathematical operation varies by the type of activation function, for example the ReLU function is calculated by $ReLU(X) = \max(0, X)$, but they all share the common goal of introducing non-linearity to the mapping of inputs and outputs in the network.

2.2.2 Unsupervised graph learning

Three unsupervised learning techniques of interest to this study when applied to graph-structured data are graph autoencoders, generative adversarial networks, and graph recurrent neural networks.

Graph autoencoders, and their extensions, variational graph autoencoders, are described by Kipf et al. [38] as trainable neural networks that are able to perform end-to-end link prediction and clustering on graphs in an unsupervised manner. Graph autoencoders consist of two models, an inference model that is tasked with encoding the input graph information, and a generative model that is tasked with attempting to regenerate the graph's structural information. In practice, GCNs could be used, among other methods, as the inference model in a graph variational autoencoder framework. It follows that functionally, graph autoencoders only differ from GCNs in that they are not used directly for classification or clustering, but rather as a dimensionality reduction technique that is able to condense a graph's node characteristics in a low-dimensional format. Such a representation can be used as the first step towards performing the aforementioned tasks, or to study the underlying relationships between graph nodes across different networks.

Operationally, in [38], graph autoencoders are given an undirected and unweighted graph G defined by its adjacency matrix A and degree matrix D. The first step in the model is to infer graph latent representation through a two-layer GCN encoder framework. The output of this step is the data's mean vector μ and standard deviation σ^2, given by:

$$\mu = GCN_{\mu}(X, A)$$

$$\log \sigma = GCN_{\sigma}(X, A)$$

where the convolutional network function is defined as:

$$GCN(X, A) = \tilde{A}\ ReLU\left(\tilde{A}XW_0\right)W_1.$$

Here, X is an $N \times D$ matrix containing the graph nodes' features, and A is the graph's adjacency matrix. For each input i, we have:

$$q(z_i|X, A) = \mathcal{N}(z_i|\mu_i, diag(\sigma_i^2))$$

where z_i represents the stochastic latent variables. The second step of the model is a decoder that attempts to reconstruct $\tilde{A}$ after getting Z. In Kipf's framework, this generative model is a simple latent variables inner product formulated as:

$$p(A|Z) = \prod_{i=1}^{N} \prod_{j=1}^{N} p\left(A_{ij}|z_i, z_j\right),$$

$$p\left(A_{ij} = 1|z_i, z_j\right) = \sigma(z_i^T\ z_j)$$

where σ is the sigmoid activation function, defined as $\sigma(X) = 1/(1 + e^{-X})$.

Generative adversarial networks (GANs) are another powerful unsupervised generative method. Originally developed by Goodfellow et al. [39], GANs have gained traction in recent years due to their impressive performance on various image applications [40]. The basic idea behind GANs is to train two neural networks, a generative network that learns from input data and attempts to create new data similar to it, and a discriminator network that is tasked with

distinguishing between original data and synthetic data created by the generator. After many rounds of training, and with the proper training parameters, the generative network continues to improve at creating new data points. The model training ends when the discriminator experiences a sufficient amount of difficulty differentiating between the real and artificial data. As with many deep learning frameworks, the choice of neural networks making up the generator and discriminator in GANs can be as simple as two multi-layer perceptron networks, as in the original Goodfellow implementation, and as large as deep, multi-layer convolutional networks, as in Karras et al.'s architecture [41].

In practice, the generator net in a GAN is tasked with mapping input noise variables z over data x and generating new data instances θ according to:

$$x_n = G(z_n, \theta_g);\ z_n \sim N(0,1).$$

The discriminator, on the other hand, takes as input both the real and generated data and determines the probability of the input x originating from the real data rather than the generator's output as:

$$D(x, \theta_d) \in [0, 1].$$

It follows that the learning aspect in a framework of this structure ought to aim towards maximizing the expectation of the discriminator prediction being correct and minimizing the instances where the generator's output is divergent enough from original data distribution. Indeed, mathematically, the generator-discriminator dynamic in a GAN can be perfectly described as a two-stage minimax optimization problem with the objective function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{z \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log\left(1 - D\left(G(z)\right)\right)],$$

where $p_z(z)$ is a defined prior on the input noise values.

The third and final form of unsupervised learning methods of interest is graph recurrent neural networks (GRNNs). Recurrent networks are a type of neural networks that are able to

represent temporal sequences through modeling connections between the network nodes as a directed graph. For this reason, RNNs have grown immensely in popularity in recent years as they evolved to be the standard in applications such as machine translation [42] and speech recognition [43]. This property, however, can also be useful when applied to the problem of graph generation, where the addition of new nodes and edges to a graph can be modeled as a temporal sequence.

Ma et al. [44] propose using a long short-term memory (LSTM), a variant of recurrent neural nets with feedback connections between its layers, in order to model dynamic graphs' node representations. During the generation process, after creating each new edge connecting two nodes, the LSTM is tasked with updating both nodes as well as neighboring nodes' representations. Their approach, while experimental, shows that a time-sensitive LSTM can be adept at learning from dynamic graph data and constructing an edge-formation schedule consisting of the time intervals and the order in which edges should be added to the graph.

In a more general direction, You et al. [45] propose an end-to-end GRNN framework for modeling the complex dependencies in graphs of various types and generating new ones based on the learned representations. The framework is simply composed of two recurrent networks, one for creating new nodes, and the other for creating edges that connect the generated nodes in a sequential manner. Unlike traditional rule-based generative methods, the authors show that GRNN-based approaches are able to learn from input graph data without significantly increasing the algorithm's time complexity. The idea of their approach is to use breadth-first-search (BFS) to linearize the structure of a given graph into a sequence of nodes, much like a sentence, that is then used to train the node-level RNN. This operation is represented in the equation:

$$S^{\pi} = f_S\big(G, BFS(G, \pi)\big),$$

where S^{π} is the set of all graph node adjacency matrices under node order π. The edge-level RNN

uses a conditional Bernoulli distribution to model the binary edge generation sequences during its training. This is done in order to fully model the convoluted edge dependencies in functional architecture graphs, and is accomplished mathematically through the decomposition of a given node's proposed connections into a product of probabilities:

$$p(S_i^{\pi} | S_{<i}^{\pi}) = \prod_{j=1}^{i-1} p\left(S_{i,j}^{\pi} \middle| S_{i,<J}^{\pi}, S_{<i}^{\pi}\right).$$

In the above equation, the edges that a given node i is connected to at every time step is determined through calculating the probabilities of it being connected to all other nodes j given the edge connections existing so far. As the size of the graph grows, so does the number of variables that are taken into consideration for future edge generation.

2.3 SUMMARY

Having examined a number of network optimization techniques and methodologies in Section 2.1, and a background on semi-supervised and unsupervised machine learning algorithms in Section 2.2, we now go into detail on our two problems of interest in Chapters 3 and 4. Chapter 3 details the usage of a multi-level optimization model to analyze various aspects of the air network resilience, and Chapter 4 presents the unsupervised detection of design patterns in architecture graphs using multiple machine learning methods.

CHAPTER 3: RESILIENCE ANALYSIS OF THE US AIR TRANSPORTATION NETWORK

Assessing and improving infrastructure resilience has been an important national security objective for the past twenty years. The US air transportation infrastructure plays a prominent role in national economic development and citizen mobility, and therefore is of special interest to this objective. This chapter presents the use of a defender-attacker-defender model to analyze potential impacts of worst-case disruptions on the US air transportation network, as well as possible protection steps that could be taken to minimize the negative outcomes of such disruptions. *(Some materials and figures included in this chapter have been published in [47] © 2019 IEEE.)*

3.1 MODELING APPROACH

The models and applications examined in Section 2.1, and specifically Subsection 2.1.3, have shown that tri-level defender-attacker-defender (DAD) models are able to provide helpful insight into the resilience of multiple infrastructure systems that are composed primarily of physical assets (e.g. road segments, bridges, electric transmission lines, weapon systems, etc.). We extend this work through the adaptation of the DAD model approach to assess the resilience of the US air transportation system on the core assumption that air routes, not airports, are the most critical components in the network, and that they can be regarded as *physical* components that can be disrupted and protected for the interest of resilience assessment. In this framework, passenger supply and demand at each airport is used to drive traffic on routes under assumed constraints of capacity and passenger preference. The justification for making such an assumption is threefold. First, due to the unique nature of the air network and the laws of competition and profit, the busiest

air routes, which also happen to be the ones most susceptible to disruption, could largely be considered to be fixed over the long term. Secondly, while airport-wide disruptions undoubtedly affect a larger number of travelers, route-specific disruptions are much more common in terms of occurrence and therefore carry an increased effect on overall network resilience. Lastly, as is the case for physical components, air routes operate as an interconnected network with vulnerabilities and strengths that could affect the entire system, which further supports our assumption.

Our DAD formulation is a tri-level mixed-integer optimization model. The first level represents a system operator that is tasked with routing passengers in the air network between their respective source and destination airports. Mathematically, this goal is achieved by calculating the origin-destination route for each passenger with the minimum travel cost, based on the network's current operational status. The second model level represents an attacker agent that observes the status of the network at a point in time, including supply and demand and traffic volume, and devises a disruption plan that stops all traffic on a limited set of network routes. The objective of the attacker agent is simply to increase the operational costs associated with passenger routing for the system operator through the incapacitation of key network routes. The third and final model level represents a system defender that works to limit any potential disruption in the network by defending a set of network routes in a way that makes them immune to attacks.

The origin-destination supply and demand data supplied to the model is obtained from the Bureau of Transportation Statistics' (BTS) Airline Origin and Destination Market Survey (DB1BMarket) dataset [46]. Each record in the dataset represents an airline ticket for a domestic US flight, and contains, among others, the following fields: origin and destination airport codes, flying distance, ticket price, and number of passengers. The data extracted from the dataset for this study correspond to a 10% sample of ticket data reported by airlines from Q4 2012 to Q3 2017. In

our model, we use traffic volume as a proxy for the supply and demand for passengers at each airport, and pricing data as a proxy for the operational cost to transport passengers on each route.

Operationally, through sequential attack and defense dynamics, the model is designed to identify the most critical routes in the air network with the highest frequency of attack/defense utilization. Such knowledge could be key to decision makers in their effort to quantify and improve the resilience of the air network. Projects involving the design and planning of new transportation modes, such as high-speed rail and on-demand unmanned aerial vehicles, stand to benefit from the knowledge of the specific air routes in most need of augmentation. Additionally, through the incorporation of the US highway network into the model, further insights could be drawn towards the identification of ground roads that could be utilized to optimally reroute passengers after the occurrence of certain costly disruption scenarios in the air network.

3.2 PROBLEM FORMULATION

The air network resilience assessment problem takes the form of a tri-level, non-linear optimization program with the following objective and constraints [47]:

$$\min_{D} \max_{A} \min_{P,R} \sum_{[i,j]\in E} \left[t_{ij} + h_{ij}A_{ij}\left(1 - D_{ij}\right)\right] P_{ij} + \sum_{n,q\in N} l_{qn} R_{qn} \tag{1}$$

$$\text{s.t.} \sum_{[i,j]\in E} A_{ij} \leq AB \qquad \forall [i,j] \in E \tag{2}$$

$$\sum_{[i,j]\in E} D_{ij} \leq DB \qquad \forall [i,j] \in E \tag{3}$$

$$\sum_{(n,j)\in Z} P_{qnj} - \sum_{(i,n)\in Z} P_{qin} - R_{qn} \leq m_{qn} \quad \forall n, q \in N \tag{4}$$

$$\sum_{q \in N} P_{qij} - P_{ij} \; = 0 \qquad \forall [i,j] \in E \tag{5}$$

$$A_{ij} \in \{0\,,1\} \qquad \forall [i,j] \in E \tag{6}$$

$$D_{ij} \in \{0\,,1\} \qquad \forall [i,j] \in E \tag{7}$$

$$R_{qn} \geq 0 \qquad \forall n,q \in N \tag{8}$$

$$0 \leq P_{ij} + P_{ji} \leq u_{ij} \qquad \forall [i,j] \in E \tag{9}$$

where $[i,j] \in E$ is an undirected route between airports i and j, $(i,j) \in Z$ is a directed route between i and j, and the indices $n,q \in N$ are airport nodes. The set N contains all network nodes, while E contains all undirected routes and Z all directed routes.

The variables and input data used in the model equations are defined in Table 1. The model's objective function in Equation (1) is a tri-level optimization problem. The first level, concerning the system operator, has the objective of minimizing the overall operational costs of routing passengers from their origin to destination points. Mathematically, the steady-state operational cost of the network can be calculated by multiplying the variable t_{ij}, representing the per-passenger cost of using undirected route $[i,j]$, and the variable P_{ij}, which represents the total number of travelers on the route. The penalty term $l_{qn}\, R_{qn}$, composed of the multiplication of the number of unrouted passengers with an arbitrary penalty constant, exists to force the system operator to route as many passengers as possible to their destinations, even in the presence of disruptions. The second level of the model, the attacker agent, is represented by the term $h_{ij}A_{ij}$, which adds a penalty to the overall objective cost for each disrupted route $[i,j]$. Aiming to increase the operational cost for the network maximally, the attacker's choice of attacked routes takes into account the routes' current traffic as well as their connections to defended routes. Adversely, the

model's third level, the defender minimization problem, only exists to mitigate the effects of attacks by the careful selection of the defended routes in variable D_{ij}. This objective is attained through the designation of one or more route values in the array D_{ij} as one, which in turn renders the term $(1 - D_{ij})$ zero for these routes; effectively preempting the effects of attacks on them.

The optimization problem's eight constraints are listed in Equations (2-9). Constraints (2-3) define the maximum number of air routes the attacker and defender are allowed to select, respectively. Constraint (4) ensures at each airport, for each airport pair, that the sum of the number of passengers arriving minus the number of passengers departing and the number of passengers who are stranded is less than or equal to the data-driven passenger supply and demand figure. Constraint (5) states that the sum of the number of passengers traveling on each route who originated from all airports must be equal to the total number of passengers on that route. Constraints (6-7) establish the arrays Aij and Dij as binary matrices where values of one indicate attacked and defended routes and values of zero indicate non-attacked and non-defended routes. Constraint (8) establishes that, at each airport for each airport pair, the number of unrouted passengers, the variable R_{qn}, must have a zero or positive value. Finally, Constraint (9) requires the total amount of passenger traffic on any given route to be smaller than an upper limit value defined by the variable u_{ij}.

3.3 MODEL DECOMPOSITION

For our mathematical model described in Equations (1-9) to be solved computationally, we represent it in code as a mixed-integer, non-linear optimization program. As has been shown in the literature review in Chapter 2, such programs often require the use of relaxations and

Table 1: Definition of model input data and decision variables [47] © 2019 IEEE.

Variable	Definition
t_{ij}	Mean airfare on a specified route $[i, j]$
h_{ij}	Penalty for traversing an attacked or non-existent route $[i, j]$
A_{ij}	Binary variable; 1 if route $[i, j] \in E$ is chosen to be attacked, 0 otherwise
D_{ij}	Binary variable; 1 if route $[i, j] \in E$ is chosen to be defended, 0 otherwise
P_{ij}	Integer variable: total number of passengers travelling on route $[i, j] \in E$
l_{qn}	Penalty of non-routed passengers at airport $q \in N$ who were originally on an itinerary ending at airport $n \in N$. In practice: $l_{qn} < h_{ij}\ \forall\ n, q \in N,\ [i, j] \in E$
R_{qn}	Integer variable: number of non-routed passengers at airport $q \in N$ who were originally on an itinerary ending at airport $n \in N$
AB	Budget constraint on the number of allowed simultaneous attacks (integer)
DB	Budget constraint on the number of allowed simultaneous defenses (integer)
P_{qij}	Integer variable: number of passengers travelling on route $[i, j] \in E$ who originated at airport $q \in N$
m_{qn}	Total demand rate at airport $n \in N$ for passengers originating at airport $q \in N$. Total supply rate of passengers originating at airport q is represented by m_{qq}
u_{ij}	Upper bound on total number of passengers on a specified route $[i, j] \in E$

approximations to reach a solution in an efficient run time. Some of the methods that researchers have historically used to solve non-linear programs are decomposition [48], [49] and implicit enumeration [50], among other techniques. Our model uses a modified form of a decomposition algorithm [47] that was first developed by Alderson et al. [51]. Its full sequence of operations is described as follows:

1. *Initialization*: a) Input DB1BMarket data and optimality tolerance $\varepsilon > 0$.

   b) Set the lower bound (LB) to $-\infty$ and the upper bound (UB) to ∞.

c) Set loop counter K to 1.

d) Set initial defense plan ${D_{ij}}^{(0)} = 0$.

2. *Attack Subproblem*: a) Given defense plan ${D_{ij}}^{(K)}$, compute attack plan ${A_{ij}}^{(K)}$ and rerouted traffic ${P_{ij}}^{(K)}$ such $z_{AS}^{UP} - z_{AS}^{LO} \leq \varepsilon\, z_{AS}^{LO}$.

b) If $z_{AS}^{UP} <$ UB, update UB to z_{AS}^{UP}.

c) If any previous attack is repeated, add the temporary constraint $\sum {A_{ij}}^{(K)} - {A_{ij}}^{(K-1)} \geq 1$.

d) If UB – LB $\leq \varepsilon$ LB, end the program.

3. *Defense Subproblem*: a) Compute defense plan ${D_{ij}}^{(K+1)}$, such that $z_{DS}^{UP} - z_{DS}^{LO} \leq \varepsilon\, z_{DS}^{LO}$.

b) If $z_{DS}^{LO} >$ LB, update LB to z_{DS}^{LO}.

c) If UB – LB $\leq \varepsilon$ LB, end the program.

4. *Looping*: a) Update loop counter $K = K + 1$.

b) Return to step 2a.

In the initialization phase, the input data is fed into the program, and the model's initial state is set. The optimality bounds are initialized to be fully permissible and the initial defense plan is set to null. Afterwards, the program calculates the optimal attack plan and rerouting strategy given the current defense plan. If the value of the objective function after this step is smaller than the upper bound, the value becomes the new upper bound. To prevent repetitive attack plans in this step, an additional measure is taken where in every new iteration if a previous attack is

repeated, the program is forced to recompute a different plan so that cycling may not occur. Next, in the defense subproblem, a new defense plan is calculated based on the updated routing decisions and attack plan, and the optimization's lower bound is taken to be the larger of the current bound and the model's objective value. The attack and defense subproblems are repeated until the difference between the upper and lower bound reaches or exceeds a value determined by the user-defined optimality tolerance ε.

3.4 DATA PROCESSING

In our model, we use the Airline Origin and Destination Survey (DB1B) database, issued by BTS, to guide the model's passenger supply and demand inputs as well as the network's operational cost. This dataset however only contains a 10% sample of airline tickets sold by participating carriers for domestic flights in the United States. In an attempt to maximize the model's utility in estimating some of the air network's realistic operational parameters, a need arises for the model to reflect actual air travel volumes across the network. The raw data obtained from DB1B contains ticket records for 240.6 million passengers reported by airlines from Q4 2012 to Q3 2017. Assuming uniform travel patterns throughout the year, the number of daily average traveler tickets included in the dataset is approximately 131.8 thousand. According to the Federal Aviation Administration (FAA), the average number of daily domestic travelers in the US in 2018 is estimated to be approximately 2.587 million [52]. The US International Air Passenger and Freight Statistics report, issued by the Department of Transportation [53], estimates the yearly number of passengers traveling internationally to and from the US to be 217.3 million in 2017. Using the same assumption of uniform average traffic, the number of passengers travelling internationally on a daily basis can be calculated to be approximately 595.3 thousand. Using the

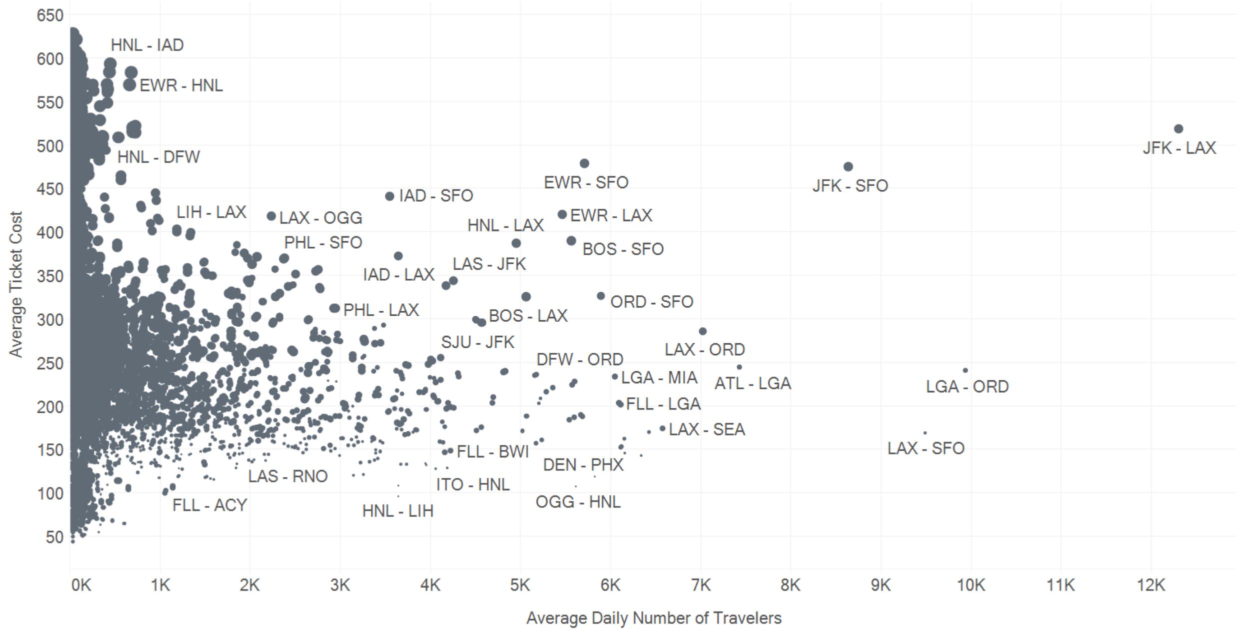

Figure 4: Distribution of the number of daily travelers and ticket costs across all domestic US air routes. Circle sizes represent relative route lengths [47] © 2019 IEEE.

total and international travel figures we've obtained so far, a rough estimate for the average daily domestic travel volume can be calculated from the two figures to be 1.992 million passengers per day. To incorporate this figure in the DB1B dataset, we use linear interpolation such that all traffic data points are multiplied by a factor of 1.992/0.1318=15.11. This gives us the estimate we need for the number of daily domestic passengers travelling on US air routes, shown in Figure 4.

The figure plots the average daily traffic versus the average ticket price for 15,469 domestic US routes. During data processing, routes with less than 15,000 average annual passengers were eliminated from the study in order to focus on the characteristics of more active routes. After elimination, the number of airports represented in the dataset is 327, comprising in whole of commercial passenger airports. In the plot, we notice a pattern of some select routes that are on the extreme ends of the data distribution, such as the high-cost, high-demand JFK-LAX and JFK-

SFO routes, and low-cost, high-demand LGA-ORD and LAX-SFO routes. In the next section, we show that the results of the optimization model indicate that these routes could potentially be disproportionately more critical to the resilience of the air network than others.

3.5 MODEL RESULTS

Having established our methodology and data sources, this section contains some operational results of the model's application to the US air network, as well as some insights that are drawn from them. Different disruption scenarios in the network are obtained through varying the maximum allowable attack and defense budgets; two integer variables that determine the number of routes the attacker and defender agents, respectively, can select in a given model run. To demonstrate the model's capabilities, we select 25 operational cases, where the attack and defense budgets vary from one to five, for analysis. As with many applications, the quality of predictions increases with the number of different run permutations, and we find that our selection of 25 run cases is adequate to obtain clear data patterns without being too computationally expensive. Every model scenario takes less than 20 minutes to reach an optimal solution on a second-generation Intel i7 processor.

Of the 25 operational run cases, we present three of them in more detail to highlight their utility in informing high-level infrastructural policy planning. The first case involves a run scenario where the attack budget outweighs the defense budget by a ratio of five to one. This hypothetical scenario is meant to explore situations where multiple disruptions occur simultaneously in the air network, as is possible during hurricanes and other natural disasters. The model's simulated attacked and defended routes are shown in Figure 5. Also shown in the figure in light blue the

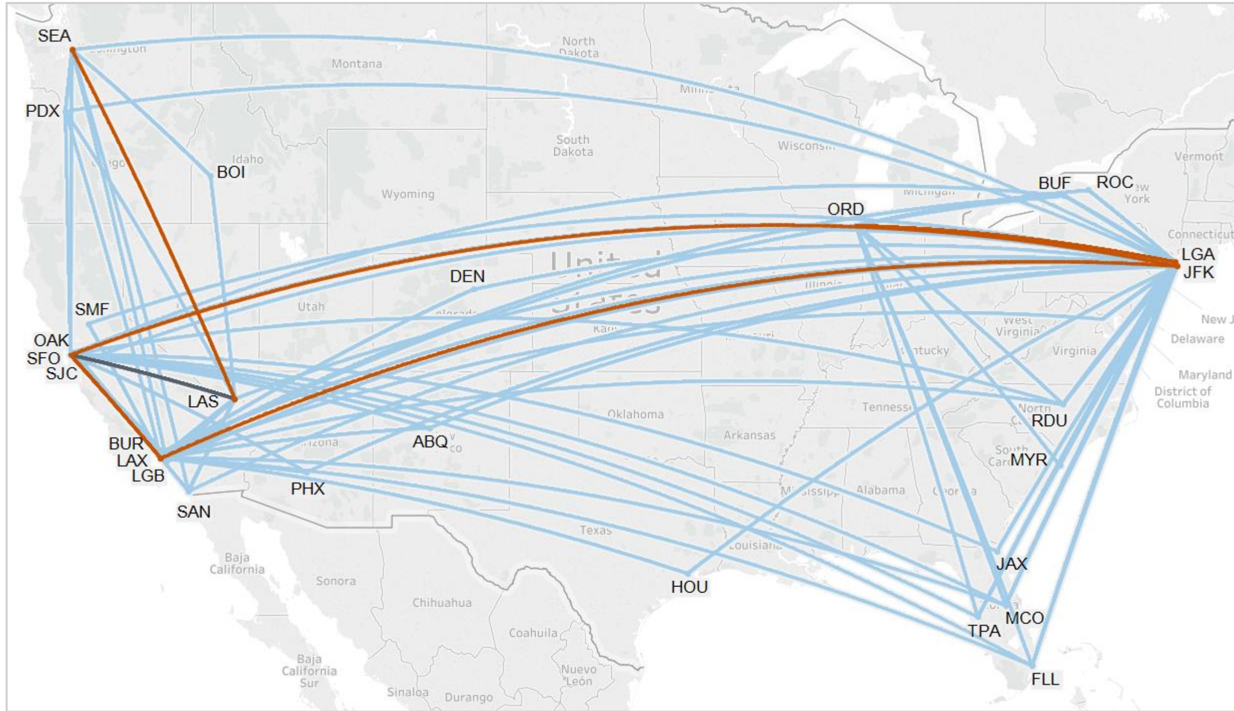

Figure 5: Best defended (Black) and attacked (Red) routes for an attack budget of five and a defense budget of one, with routes used for optimal passenger rerouting (Blue) [47] © 2019 IEEE.

routes that could be used for optimal rerouting of the passengers during and after the disruptions, until all routes return to full operability. In this case, the model determines that the five routes that would most negatively impact the air network's operations when disrupted simultaneously are the ones shown in red. The impacted routes are notably critical to inter-coast connectivity, and as evident by the high number of required emergency rerouting operations, ensuring their health is important to the overall network resilience. This underscores the need for alternative high-speed transportation options to augment the air network and minimize the effects of such impacts. of required emergency rerouting operations, ensuring their health is important to the overall network resilience. This underscores the need for alternative high-speed transportation option

The second operational scenario we are interested in explores the case where the allowable defense budget is five times that of the attack budget. This scenario would emerge realistically if

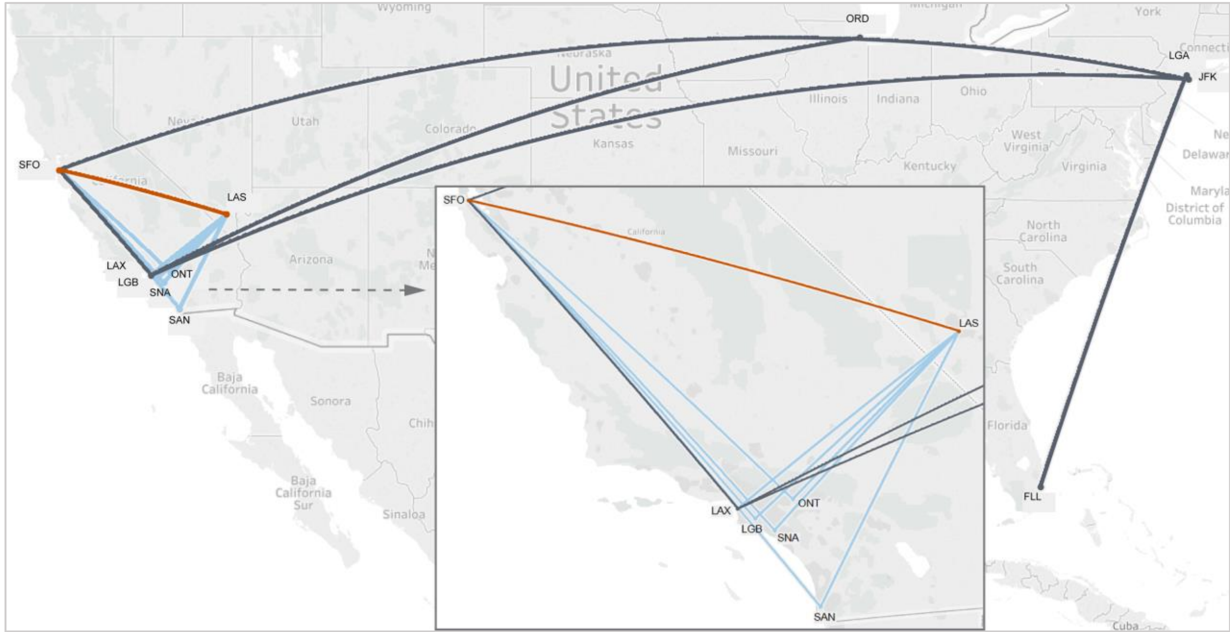

Figure 6: Best defended (Black) and attacked (Red) routes for an attack budget of one and a defense budget of five, with routes used for optimal passenger rerouting (Blue) [47] © 2019 IEEE.

an alternative mode of transportation was constructed to augment select air routes, and a disruption occurs and prevents traffic on an air route that is not augmented. Figure 6 shows the simulated optimal attacked, defended, and rerouting routes. As we examine the results of this scenario, we notice an opposite trend to the first one; as more long-range, inter-coast routes are augmented, the potential impact of any one disruption diminishes significantly. In this case, the number of overall disrupted passengers is a sixth of that of the first case. The per-passenger rerouting cost associated with post-event recovery is also reduced by approximately 14%. The far fewer routes needed to reroute passengers compared to the first case is an indicator of the much reduced impact of individual disruptions even on a partially-augmented air network. In other words, although building a new nation-wide transportation network could be arduous, strategically building individual lines to augment critical air routes could be a more affordable and beneficial endeavor.

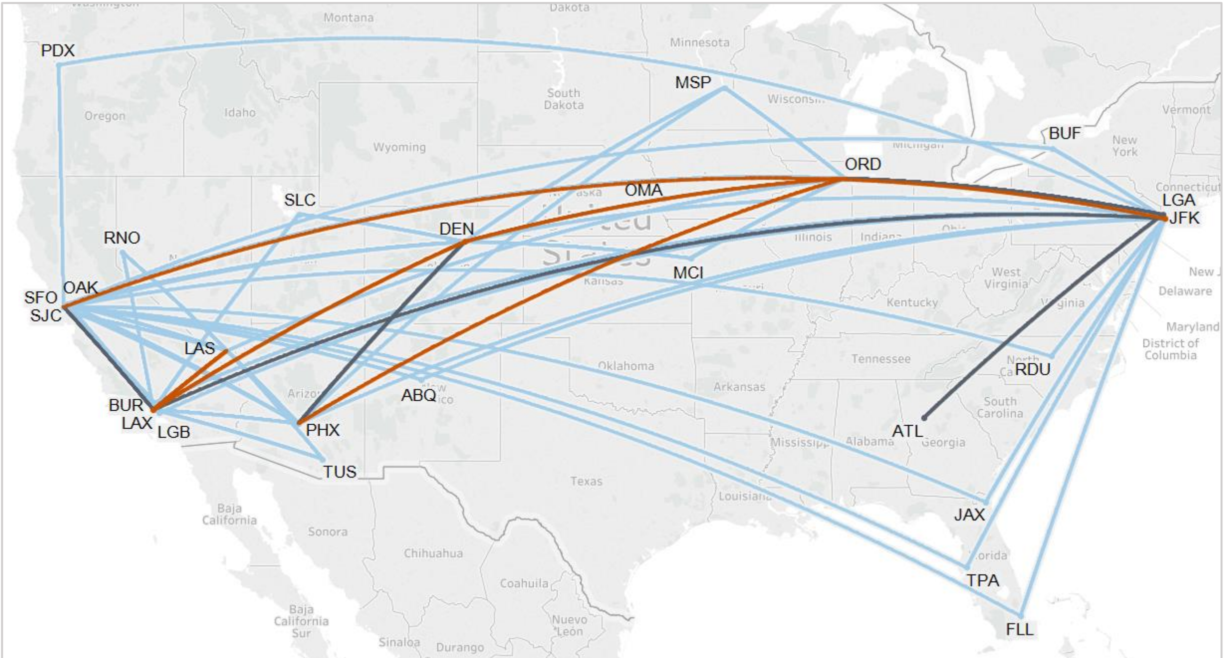

Figure 7: Best defended (Black) and attacked (Red) routes for attack and defense budgets of five, with routes used for optimal passenger rerouting (Blue) [47] © 2019 IEEE.

The final operational scenario we examine closely involves having an equally large attack and defense budgets of five as input to the model. This is a combination of the first two scenarios, in which both the air network is partially-augmented and multiple disruption events are occurring simultaneously. Figure 7 contains the model's optimized selections for attacked and defended routes, and those used for rerouting. Compared to the first case, with the same attack budget, we see a decrease of 38% in the number of disrupted travelers overall. The per-passenger cost of rerouting also drops by approximately 9% between the two cases. This further reaffirms our position that even a modest augmentation to the air network along some carefully selected routes could potentially increase the air network resilience significantly, without requiring the same amount of resources that a larger-scale project would need. To summarize the three operational cases, Table 2 contains exact numerical results from each of them and how they compare.

Table 2: Summary of the three operational test cases [47] © 2019 IEEE.

Attack budget	Defense budget	Number of disrupted passengers	Per-passenger rerouting cost ($)
1	5	7,246	713.26
5	1	44,076	813.61
5	5	27,608	744.37

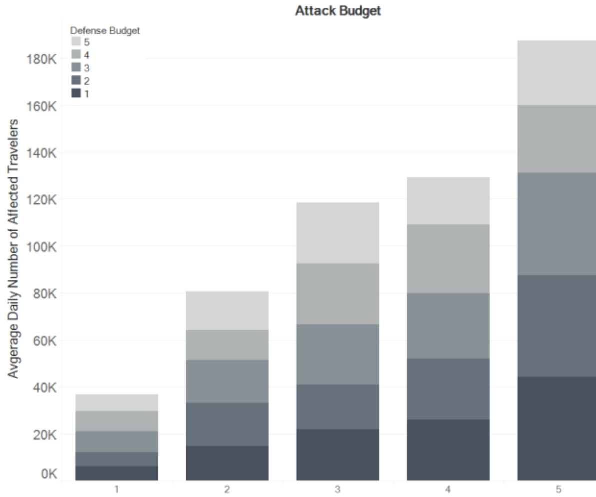

Figure 8: Estimated number of daily affected passengers for attack and defense budgets ranging from one to five [47] © 2019 IEEE.

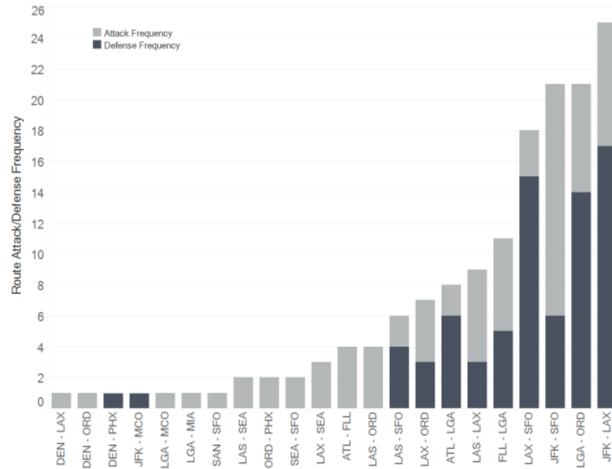

Figure 9: Most frequently attacked and defended routes based on twenty-five test cases representing attack and defense budgets ranging from one to five [47] © 2019 IEEE.

Going beyond the three operational cases, we now examine some insights that are drawn by analyzing the optimization metrics from all 25 run cases collectively. We are interested in studying how the number of disrupted passengers change in every scenario, and the possible effects network augmentation could play to impact that metric, and by proxy, the network resilience. Figure 8 plots the average daily number of disrupted travelers for all operational cases against both model attack and defense budgets. Non surprisingly, this number increases with the increase in attack budgets, but as we have noted before in the three operational cases analysis, we can see a clear trend towards the decline of the number of disrupted passengers as the defense budget

increases in most cases. We note however that in some cases, the number of affected passengers increases instead with the increase of defense budget. We postulate that this is resultant of cases where affordable and short alternative air routes are readily available for the passengers, and it is deemed by the model that a less-connected route that is comparably central to network connectivity and is in more need of augmentation.

One key metric we can use to determine the relative criticalness of different routes in the air network is the frequency of their inclusion in the model's attacked and defended route sets. Figure 9 is a visualization of this information. Among the 15,469 routes included in the source data, we notice that only 22 routes have been chosen at least once by the model, with the top four being selected more often than the bottom 18 combined. Looking back at the traffic and pricing data shown in Figure 4, we note that the same four routes are generally characterized by high ticket cost and average daily traffic. It follows that from an operational standpoint, ensuring the continuous functioning of these routes, with augmentation as one possible way to do this, could be vital to maintaining the air network resilience. Moreover, we note that many of the top routes on the frequency chart are medium to long-range routes connecting the two coasts, or either coast and the Midwest. This signals that for new transportation infrastructure projects, maintaining inter-coast connectivity could potentially be an important resilience metric to consider in the planning of new network routes.

Another study element of interest is identifying the specific routes that could be used during and after a given disruption to reroute as many passengers as possible to their original destinations. Such routes, of course, vary based on the location and number of simultaneously disrupted routes. By aggregating the data from all 25 run cases, however, we are able to get a general idea of the routes that the model deems repeatedly useful in post-disruption recovery, which could serve as

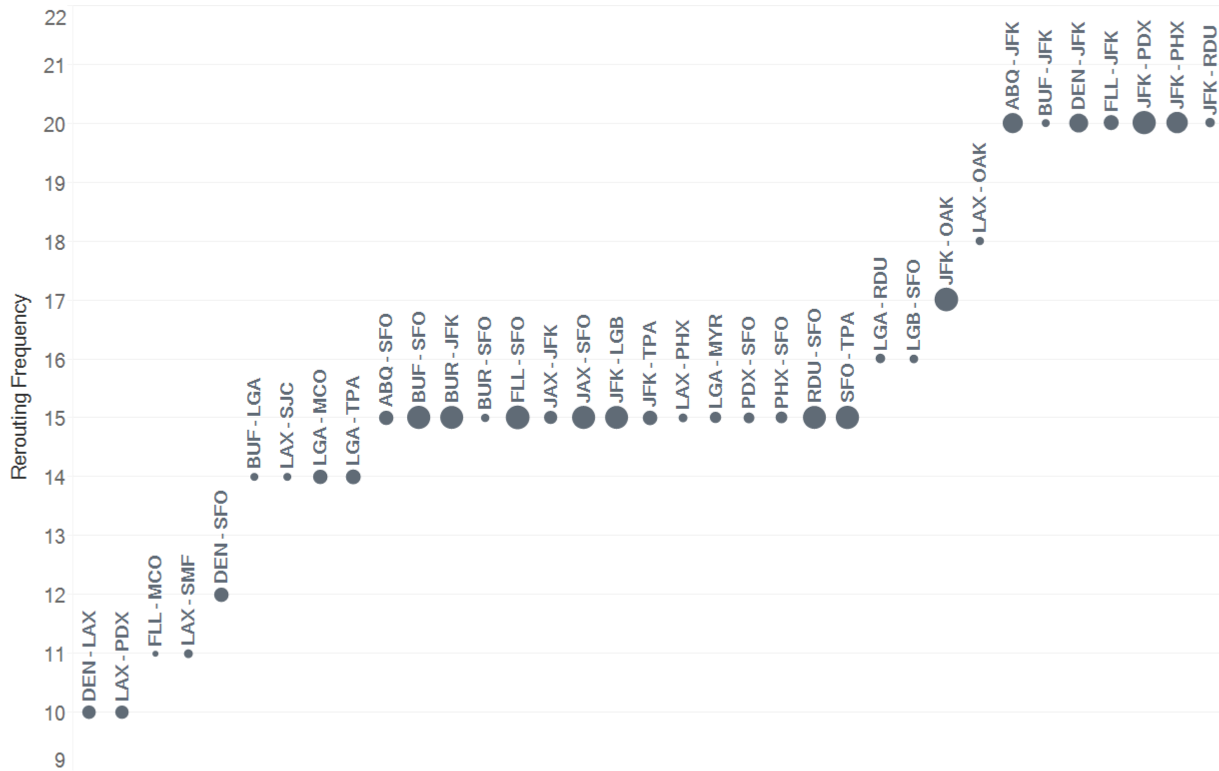

Figure 10: Most frequently used routes for rerouting operations following one or more attacks. Circle sizes represent relative route lengths [47] © 2019 IEEE.

an indication of their importance to network connectivity. The routes and the frequency of their inclusion in the model rerouting plans are shown in Figure 10. Examining the data closely, we note the top seven routes are all routes connecting the east coast to other regions. Individually, they have each been needed in 20 out of 25 operational cases for the rerouting effort. This further strengthens the notion that a strong and diverse inter-coast transportation systems could be key to the resilience of the transportation infrastructure in general, and the air network in particular.

Our results show that network optimization has the potential to uncover valuable insights into the resilience of the air transportation network in particular, and the greater transportation infrastructure in general. Future work directions could include the incorporation of schedules and delays as a temporal component to the problem, and the addition of more refined data sources.

CHAPTER 4: UNSUPERVISED PATTERN IDENTIFICATION IN ARCHITECTURE GRAPHS

Design patterns have been considered increasingly important in the software industry as they enable programmers to find proven solutions for recurring challenges and provide a way to define better code-writing practices and quality assurance. The discovery of these design patterns in software repositories is an important step towards a deeper understanding of the commonalities between different software frameworks, and by extension, a reference to help guide the engineering of future systems. Traditionally, this discovery process has been performed using various supervised and semi-supervised methods. Some are heuristic in nature, and others leverage various forms of optimization and machine learning. In this chapter, we propose a model to automatically generate realistic software architectures, as well as discover the patterns in the generated architectures, in an unsupervised manner using machine learning. We compare our generation results to the original data, and show that the discovered design patterns are similar in form and function to established industry patterns.

4.1 PROBLEM SUMMARY

In the past 20 years, the commercial market has seen an influx of software architectures, aided by the near-unlimited resources and distribution potential of the internet. As the software systems grew in size and scope over time, so did their complexity and inter-connectivity, and with them the need for better analysis and development tools. For this reason, design patterns emerged in the mid-nineties as a much-needed systematic method of defining reusable elements in object-

oriented programming [54]. Well known design patterns that are often used in software systems include the Adapter, Composite, Bridge, Façade, Command, and Proxy patterns [55].

As these patterns are mature and well-defined, most design pattern identification approaches in the literature are rule-based and/or supervised in nature. That is, they rely on pre-existing knowledge of pattern graph structure, which in turn enables the problem to be reduced to finding the most optimal way to match this structure to a corresponding sub-structure in a given architecture. Researchers, as early as 1996 [56], have been proposing different algorithms, with varying pattern identification capabilities, to search for patterns embedded in Java and C++ software systems. Some algorithms provide exact matches between patterns and system graphs [57], [58], while others are only able to provide approximations [59], [60]. Some approaches leverage matrix format for architecture and pattern graph representations [61], [62], while others use standard computer science representations such as abstract syntax trees (AST) [63] and abstract semantic graphs (ASG) [64]. They all, however, require partial or full knowledge of certain patterns of interest, and are only able to identify those same patterns in system architectures.

Going beyond seeking to identify known design patterns in software systems, the aim of this study is to discover mostly unknown patterns in architecture graphs representing operational systems, not limited to software, in a completely unsupervised manner using deep learning. Given a dataset of system architectures in graph form, our model's goal is to recognize recurring patterns in graphs and to extract them for further analysis. Practically, there are a few obstacles in the way of achieving this goal. The most critical of which is the lack of publicly available, realistic architecture graphs depicting complex non-software systems, such as aircraft and mechanical machinery. As many deep learning approaches often require a large amount of training data, the choice of model implementation naturally gets restricted due to the data shortage.

One possible solution to the lack of training data is simply to generate realistic data points, in this case system architectures, via techniques that require little to no input data. Traditional methods of generating viable network graphs include, but are not limited to, the Barabási–Albert model [65], mixed membership stochastic block models [66], Watts-Strogatz graphs [67], and exponential random graph models [68]. All of the listed models are specialized, non-general models that are carefully designed to approximate the distribution of only certain types of graphs. For example, the Watts-Strogatz graphs are engineered to mimic the dynamics of small-world networks such as those of Caenorhabditis elegans (roundworms), while methods such as exponential random graphs are more suited towards the modeling of social networks. Although very powerful in their respective domains, the abovementioned models' biggest drawback is their inability to *learn* the distribution of new types of graphs given input data, only relying on pre-coded knowledge that is compiled by subject-matter experts.

New advancements in machine learning have started to fill the need for models that are able to learn and adapt to different forms of network data. As covered in Chapter 2, methods such as variational graph autoencoders (VGAEs), generative adversarial networks (GANs), and graph recurrent neural networks (GRNNs) have proven highly capable of generating realistic network graphs with varying degrees of training data requirement. For this study, we have chosen recurrent neural networks as the model of choice for generating architecture graphs, the reason of which is two-fold. First, due to their autoregressive nature, and through modeling graphs as sequences, GRNNs are able to naturally capture the complex dependencies between graph nodes and edges. Second, unlike VGAE-based methods, recurrent networks are able to generalize to variable sized graphs in their output space in a way that enables the generated networks to not be limited to a fixed number of nodes.

The next step after the formation of our dataset through graph generation is to filter the created graphs based on their structural properties in order to only keep those that are within a certain degree of similarity to the original model graph. Following this operation, a machine learning classifier is trained on the original graph node and edge types, and is used to predict the same properties for the generated graphs after filtration. Next, we embed the labeled graphs into a low dimensional state space using a graph autoencoder. This enables us to obtain a concise representation of the generated data, in an unsupervised manner, that contains information about graph node inter-dependencies and how certain node types tend to be more connected. The final step in the process is to leverage the information learned through graph embedding to cluster the generated graphs into subgraphs composed of nodes and edges that are strongly related, using unsupervised hierarchical clustering. By examining the structure and properties of obtained subgraphs, we finally group subgraphs by frequency of occurrence, and weigh whether the most frequently appearing subgraphs could be considered as design patterns.

4.2 SOLUTION METHODOLOGY

Our approach to solving the pattern identification problem starts with generating enough graph data for use in later embedding and clustering stages. To this end, we adopt You et al.'s generative algorithm [45] with some modifications for it to suit our problem. The main idea behind using recurrent networks to generate new graphs is that by mapping graphs as sequences and learning a function over this sequence, a task that recurrent nets do remarkably well, we are able to generate new sequences (graphs) that are very close in their distributions to the original graphs. This approach generally relies on two neural nets, one to generate graph nodes and the other to generate edges sequentially, taking into account current graph connectivity at every time step in

the process. In our implementation, the first network is a gated recurrent unit (GRU) that is tasked with generating new graph nodes and maintaining the state of the graph at each time step. The second network in our modified model is not recurrent, but rather a simple multi-layer perceptron (MLP). The MLP accepts the output of the first network as input and is tasked with generating a probability distribution for the creation of new edges in the graph. We find that, due to our limited available training data, a shallow, non-recurrent network in the second step can help improve the performance of the model and reduce overfitting. Additionally, we modify the framework's graph encoding and decoding mechanisms, that are designed to only work with undirected graphs, to allow it to process directed graphs as well.

Operationally, the node-level GRU takes as input the net's hidden state of the graph after the creation of the previous node in the sequence h_{i-1}, and its adjacency matrix S_{i-1}^{π}, and outputs the new graph hidden state h_i after the addition of node i. The mathematics behind GRU operations are as follows:

$$z_i = \sigma\left(W_z \cdot [h_{i-1}, S_{i-1}^{\pi}]\right), \tag{10}$$

$$r_i = \sigma\left(W_r \cdot [h_{i-1}, S_{i-1}^{\pi}]\right), \tag{11}$$

$$\tilde{h}_i = \tanh\left(W \cdot [r_i * h_{i-1}, S_{i-1}^{\pi}]\right), \tag{12}$$

$$h_i = (1 - z_i) * h_{i-1} + z_i * \tilde{h}_i. \tag{13}$$

Here, z_i, r_i, and $\tilde{h}_i$ are called the update, reset, and current memory gates, respectively. σ and $\tanh$ refer to the sigmoid and hyperbolic tangent activation functions. W_z, W_r, and W are the weight matrices that drive network learning. A schematic of the GRU is shown in Figure 11. The edge-level MLP, on the other hand, is simply composed of two linear layers which share weights at all

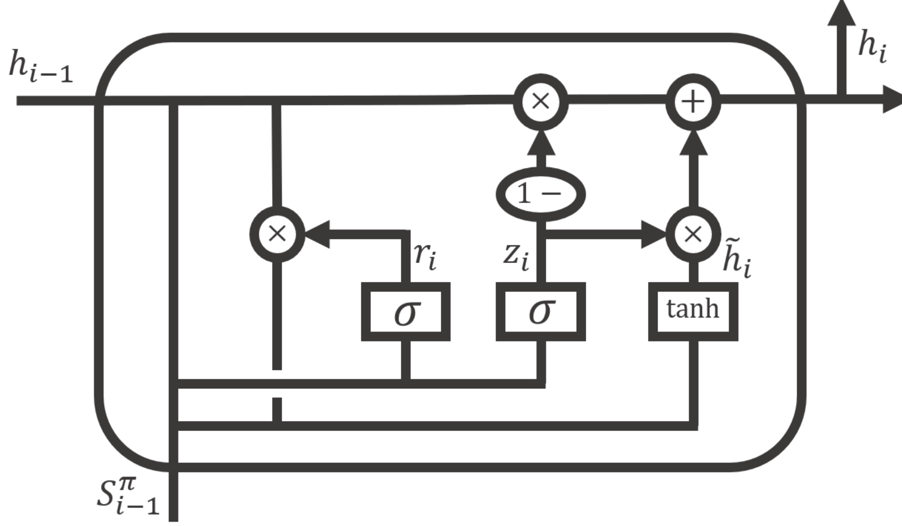

Figure 11: Schematic of the model's node-level gated recurrent unit.

training steps, with a ReLU activation layer in the middle. The input to the MLP is the hidden state of the graph generated so far h_i, and the output is the distribution of the adjacency vector of the next node in the generation sequence θ_i. The overall model, therefore, can be described as the composition of two functions, a state transition function, modeled as a GRU, and an output function in the form of an MLP:

$$h_i = GRU\ (h_{i-1}, S^{\pi}_{i-1}), \tag{14}$$

$$\theta_i = MLP(h_i). \tag{15}$$

Each element $\theta_i(j)$ in the distribution variable θ_i can be considered as the probability of an edge existing between nodes i and j. After the creation of each node in the directed graph, edges are generated through the independent sampling of S^{π}_i using a multivariate Bernoulli distribution that is parameterized by θ_i.

At inference, a generated graph's completed adjacency sequence is computed according to the following algorithm:

1. *Input*: a) State transition function GRU and output function MLP.

   b) Multivariate Bernoulli distribution $\mathcal{P}_\theta$ and the vector θ_i.

   c) Empty graph hidden state h'.

   d) Start token ST and end token ET.

2. *Preprocessing*: a) Initialize the first node's adjacency vector: $S_1^\pi = ST$.

   b) Initialize the first hidden state: $h_1 = h'$.

   c) Initialize node counter: $i = 1$.

3. *Computation*: a) Update graph state: $h_i = GRU\ (h_{i-1}, S_{i-1}^\pi)$.

   b) Update distribution vector: $\theta_i = MLP(h_i)$.

   c) Sample edge relations for node i: $S_i^\pi \sim \mathcal{P}_{\theta_i}$.

4. *Looping*: a) Update node counter $i = i + 1$.

   b) Stop if $S_i^\pi = ET$. Otherwise, return to step 3a.

5. *Output*: a) Graph sequence $S^\pi = (S_1^\pi, \dots, S_{i-1}^\pi)$.

After generation, to ensure that all the created graphs are similar, but not identical, in structure and properties to the training graphs, we evaluate each of them based on several metrics: density, average clustering coefficient, average local efficiency, radius, and diameter. Graph density is calculated by dividing the number of existing edges by the number of all possible edges. A graph's average clustering coefficient is a measure of how close each graph node's neighbors are to being a clique. A graph's average local efficiency is a measure of how well information is

exchanged by each node's neighbors when it is removed. Finally, the radius and diameter of a graph represent the minimum and maximum eccentricity, respectively, among all graph nodes. Together these parameters provide a good insight on the structure of a given graph, and how its nodes are connected. The selection criteria in terms of how a generated graph is allowed to diverge from the training data is (0.01, 0.25). In other words, we accept generated graphs that are at least 1%, and at most 25% different from training graphs in all parameters and discard the rest.

The generated graphs do not contain node nor edge labels, as the RNN architecture only learns the structural, but not functional, properties of training graphs. However, since our goal is to uncover design patterns in different types of realistic architecture graphs, and most of them do contain node and edge labels as essential components of their functions, it becomes imperative for our created graphs to contain these labels as well. As discussed in Chapter 2, classification is a task that a multitude of machine learning techniques have been developed to solve. Assuming our original training data do contain this labeling information, it becomes quite straightforward to train a machine learning classifier on these labels, and then use it to predict the node and edge labels of our created graphs. For the node classification task, we use a random forest classifier to predict node labels. Random forests are a machine learning method that works by calculating a large number of decision trees and choosing the class output based on the average prediction of all trees. In our case, the input to the node classifier at inference is a set of node properties for each node, and the output is its predicted node label. These properties are: node in-degree, out-degree, clustering coefficient, core number, in-degree centrality, out-degree centrality, closeness centrality, betweenness centrality, square clustering coefficient, and PageRank. During training, the classification model observes the values of these properties in the training data and their associated node labels, and creates a fit function that is then used to make predictions on label-less

data. For the edge classification task, we utilize support vector machines (SVMs) to make edge label predictions. SVMs work by attempting to find a hyperplane in an N-dimensional space, where N is the number of data properties, that distinctly divides the data points into different classes. The properties we use as input in this case are simply the node types of each edge's source and destination nodes. The classifier is trained using the edge labels that are present in the original data, and then used to make predictions on generated graphs after node classification is complete.

Having obtained a dataset of fully-labeled artificial graphs through generation and classification so far, the next step in the process is to encode the generated graph information into a low-dimensional space in preparation for the clustering task. To this end, we use Kipf et al's unsupervised autoencoder architecture [38], introduced in Chapter 2, to embed the generated graphs' adjacency and property matrices. During training, the model optimizes its variational lower bound $\mathcal{L}$ according to the equation:

$$\mathcal{L} = \mathbb{E}_{q(Z|X,A)}[\log p(A|Z)] - \mathrm{KL}[q(Z|X,A)||p(Z)], \quad (16)$$

where A is the graph's adjacency matrix, X is the graph's node property matrix, and Z is the latent variable matrix. The term $\mathrm{KL}[q(\cdot)||p(\cdot)]$ represents the Kullback-Leibler divergence between $q(\cdot)$ and $p(\cdot)$, a measure of how the two probability distributions differ. The distributions $q(Z|X,A)$ and $p(A|Z)$ represent the inference and generative components of the model, respectively, and are defined as follows:

$$q(Z|X,A) = \prod_{i=1}^{N} \mathcal{N}\left(z_i \middle| \mu_i, diag(\sigma_i^2)\right), \quad (17)$$

$$p(A|Z) = \prod_{i=1}^{N} \prod_{j=1}^{N} \sigma\left(z_i^T z_j\right), \quad (18)$$

where μ_i is the mean vector for node i and σ is the sigmoid activation function. Operationally, the model is able to produce useful latent embeddings of our generated graphs when given as input their adjacency, as well as property and labeling information. Such embeddings are key to analyzing connections between graph components at the subgraph level.

The last major step in our study is to perform clustering on our obtained graph embeddings using agglomerative clustering, a form of machine learning hierarchical clustering. Agglomerative clustering works by treating each data point as its own cluster initially, then successively merging them together according to a specified linking criteria. Is this case, the criteria is to minimize the sum of squared differences across all clusters. The algorithm takes as input the number of desired clusters and each generated graph's adjacency matrix, and outputs the computed clusters for each graph. After obtaining the clusters, representing the possible subgraph patterns of our generated data, we perform statistical similarity analysis based on the subgraphs' numerical properties, and group the most frequently occurring subgraphs for further analysis.

4.3 MODEL RESULTS

The first step in the graph generation process is to define the architecture training graphs that the RNN will learn from. As mentioned previously, such graphs are hard to obtain in practice for physical operational systems as, in many cases, they contain proprietary information. For this study, we use a single graph representing the high-level functions of a fighter aircraft, put together from anonymized public data. The graph, displayed in Figure 12, is fully directed and labeled, and contains 340 nodes and 500 edges in total. The graph nodes fall into one of 11 classes, and its edges one of four classes. We note that while we are only using one input graph to train the

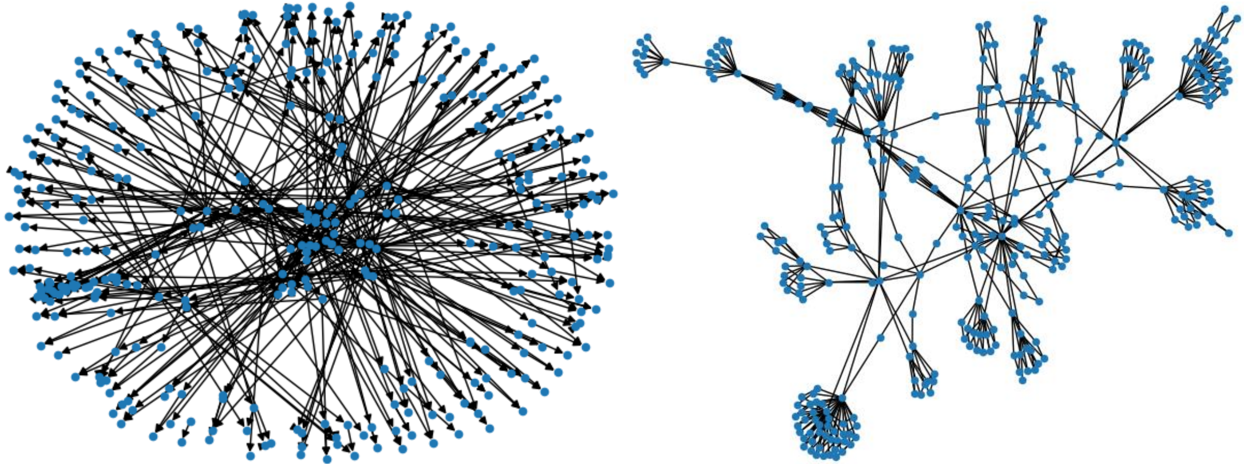

Figure 12: Visualization of our training architecture graph, in directed form (left) and undirected form (right).

generative model, our framework is entirely general and capable of processing any number and type of input graphs, if readily available.

The generative RNN model is trained using a hidden layer size of 128, an embedding output size of 64, and an initial learning rate of 0.003 for a total of 10,000 epochs. Testing and inference are performed at a 250-epoch interval, starting at the 250th epoch during training. At each inference stage, 100 graphs are generated and saved in GraphML format for later processing. The learning rate decreases by a factor of 0.3 after the 1000th and 2000th epoch marks, to eventually reach a final learning rate of 0.00027. Training lasts approximately 90 minutes when run on a P100 GPU with 32 GB of RAM.

After training the RNN for the full 10,000 epochs, we obtain 40 GraphML files, each containing 100 generated graphs sampled at different stages in training. For evaluation, we calculate each individual graph's density, average clustering coefficient, average local efficiency, radius and diameter, and only keep those that are within 1% to 25% of the architecture graph in all

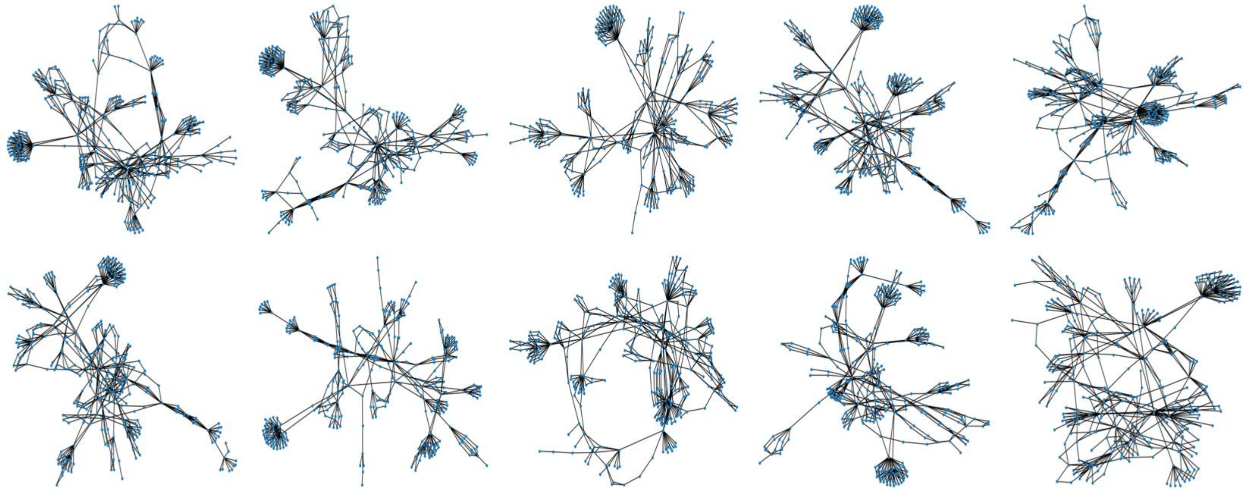

Figure 13: Visualization of a random sample of ten generated graphs (transformed to undirected form to show distribution).

properties. Of the 4,000 generated graphs, approximately 750 are found to be within this selection range in our case, just under 20% of all graphs. Figure 13 shows a visualization of a random sample of ten graphs that passed evaluation and their distributions. By the mere visual comparison of Figures 12 and 13, it is evident that the generated graphs share similar structural properties to the architecture graph, while also having unique substructures and connectivity characteristics. To examine these properties more deeply, we list those of the architecture as well as generated graphs in Table 3 and visualize them in Figure 14. We notice that just as the visual representations of the generated graphs are aligned with the architecture graph, so are their structural properties. We also note the diversity in parameter distribution between generated graphs, where no two graphs in our random sample were exceedingly close in all measured properties. This result gives us confidence that the generated graphs are realistic, yet different, enough to provide useful insight on the possible design patterns in architecture graphs. Notwithstanding, with more training data, the results of the generative algorithm could be even more realistic, and the insights more specific.

Table 3: Comparison of structural attributes of the architecture graph and a random sample of ten generated graphs.

Graph	Radius	Diameter	Average Degree	Density	Average Clustering	Local Efficiency
Architecture	6	12	2.9440	0.0087099	0.0178439	0.0195383
Generated #1	7	13	2.9612	0.0088658	0.0196491	0.0213637
Generated #2	6	12	2.9176	0.0086066	0.0162625	0.0179519
Generated #3	8	16	2.8537	0.0085441	0.0180571	0.0197716
Generated #4	5	9	2.8542	0.0097082	0.0150951	0.0150951
Generated #5	8	15	2.8690	0.0085643	0.0134306	0.0148425
Generated #6	7	13	2.9583	0.0088308	0.0169191	0.0186286
Generated #7	8	16	2.7711	0.0083718	0.0167049	0.0184350
Generated #8	8	15	2.9641	0.0089011	0.0137032	0.0154230
Generated #9	7	14	2.8810	0.0085998	0.0165061	0.0182155
Generated #10	7	11	3.0149	0.0090267	0.0174601	0.0188761

After graph evaluation, we move onto the node and edge classification segment of our process. For the node classification task, we train our random forests classifier on specific node properties extracted from the original architecture graph, and their associated node labels. These properties are: node in-degree, out-degree, clustering coefficient, core number, in-degree centrality, out-degree centrality, closeness centrality, betweenness centrality, square clustering coefficient, and PageRank. The node labels present in the architecture graph are: Class, Dependency, Extension, Operation, Package, Parameter, Port, Profile, Property, Signal, and Stereotype. The role of the classifier is to create an internal mapping function from the node properties to the node labels that can be used to make predictions on data with properties but no labels. To measure the performance of the classification model, we heuristically divide our architecture graph data into a training set and a test set with an 80/20% split, making sure that all

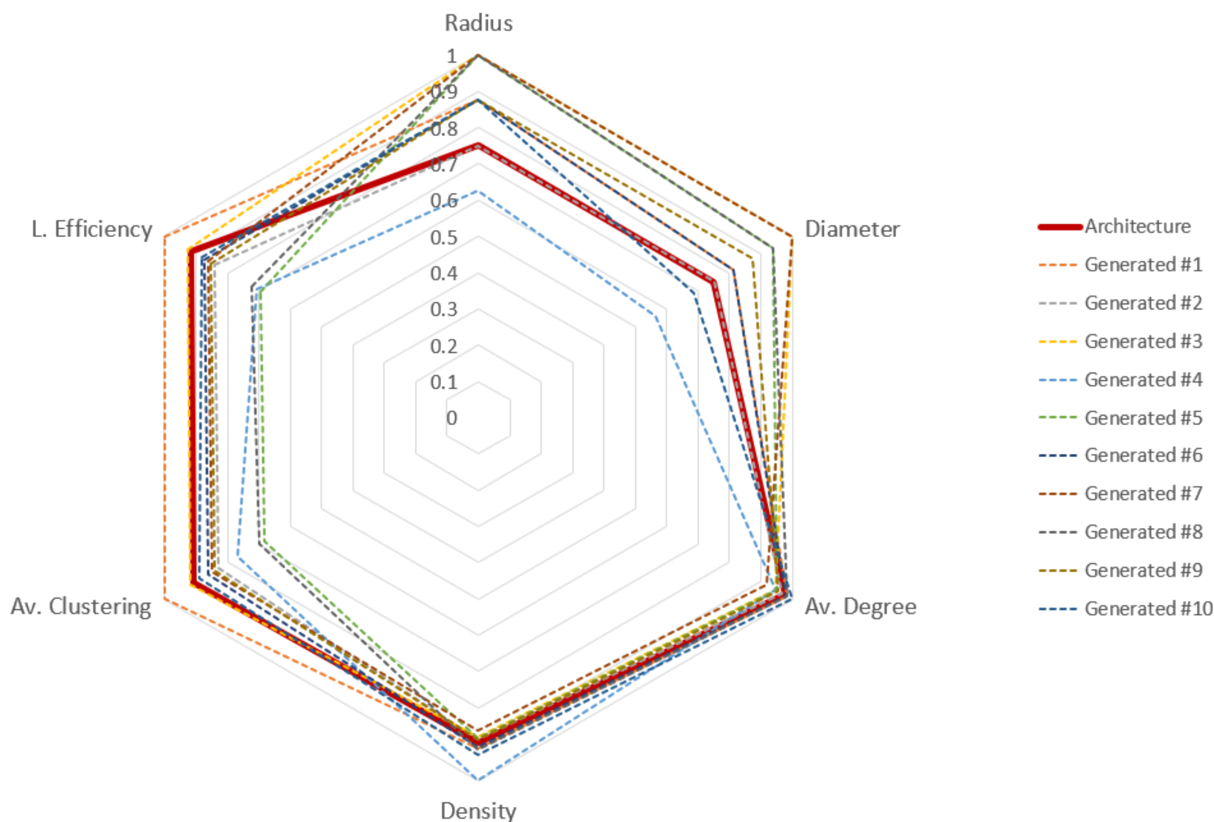

Figure 14: Visualization of the normalized structural attributes of the architecture graph versus a random sample of ten generated graphs.

11 node labels are represented in both sets. We train the model using only the training set, and then use the trained model to make predictions on the unseen test set, the results of which are shown in Figure 15. Of the 69 nodes included in the test set, the model is able to correctly predict the node labels of 65 of them, amounting to a 94.2% test accuracy. When tested on all architecture graph nodes, the model accuracy rises to 97.9%. Having validated the model performance, we finally use the trained model to predict the node labels of our 750 generated graphs.

The edge classification task, in our case, is fairly straightforward. Since there are only four edge labels present in the architecture graph, our SVM classification model is able to achieve a 100% accuracy when trained with a similar 80/20% split on the architecture graph edge data. The

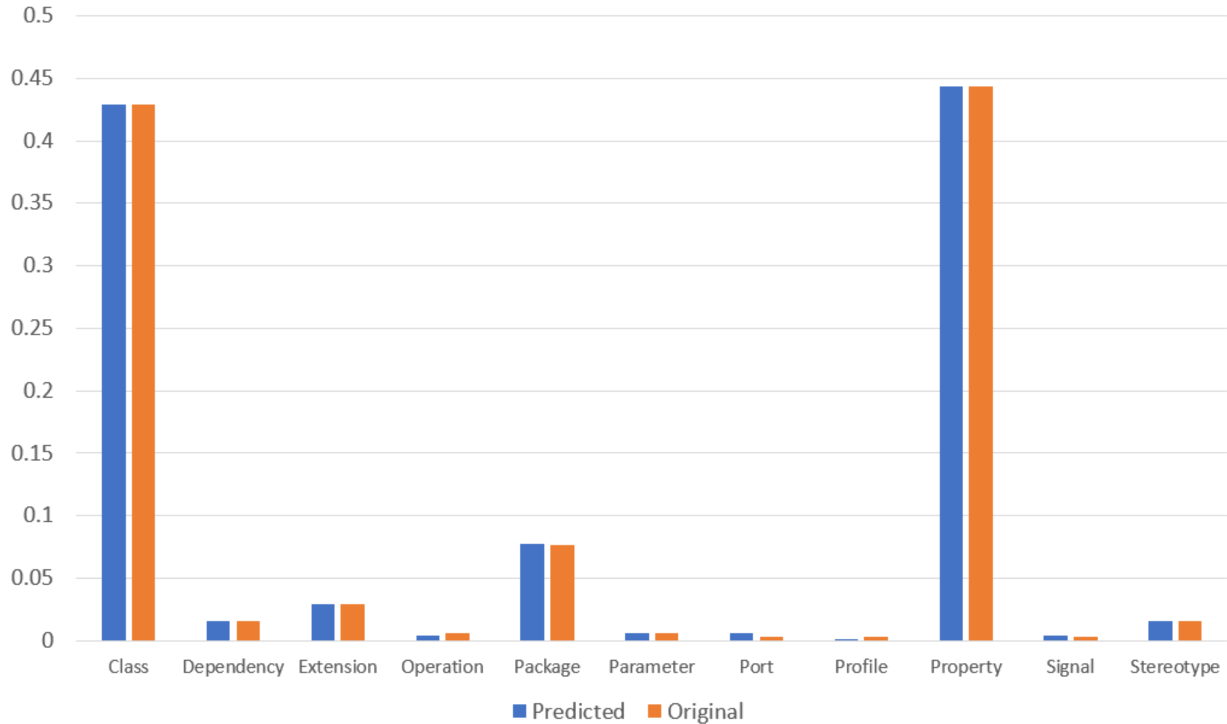

Figure 15: Node classification model test results on unseen node data.

input to the model in this case is the labels of the two nodes that the edge connects, and its output is the corresponding edge label. It is worth noting however that since the inputs to this model are reliant on the results of the node classifier at inference, we can expect the predicted edge labels on the generated graphs to contain some error that is close to the error in node labeling.

After classification, we now embed our generated graphs using the modified graph autoencoder model. For training, we set the number of epochs to 200 and the learning rate to 0.01. To determine the optimal size of the model's two hidden layers in training, we experiment with six different sizes, ranging from 32 and 16 to 8 and 4 for the size of the first and second hidden layers, respectively. The results of our experiments are displayed in Figure 16. We adopt the 32×8 latent space size for our training, as it produces a reasonably high average graph reconstruction accuracy while minimizing the variance between different test runs. The inputs to

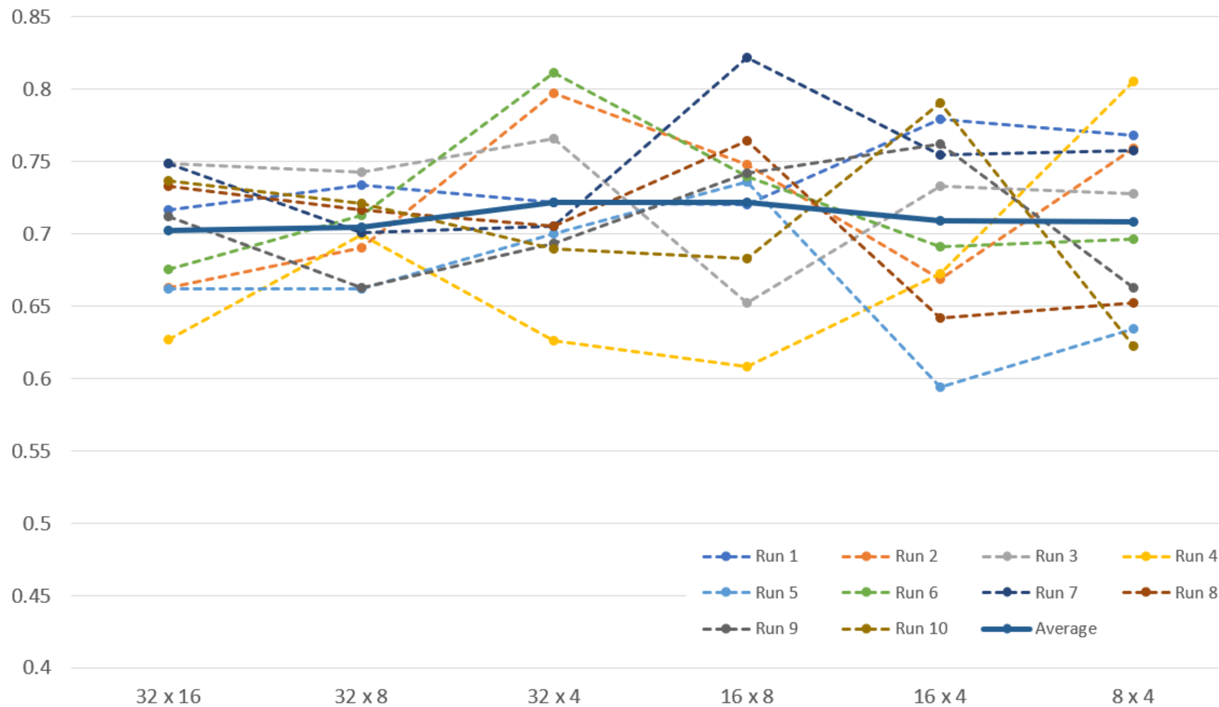

Figure 16: Reconstruction accuracy versus latent space dimensionality, as measured across ten runs of the graph autoencoder model.

the autoencoder model are each graph's adjacency matrix, as well as its property and node and edge label matrices that are encoded using one-hot encoding. The outputs of the model are each individual graph's embedding that is of size $N \times 8$, where N is the graph's number of nodes.

The last, and most important, step in our process is to extract the important subgraphs that comprise each graph in our generated graphs dataset. We accomplish this task through the use of the unsupervised agglomerative clustering algorithm. We choose this algorithm because of its ability to take into account graph connectivity, as well as node latent representations when dividing the data points into clusters. The inputs to the algorithm are the node embeddings for each graph in the dataset as well as its adjacency matrix, and the outputs are the subgraphs that make up each generated graph. After obtaining all such subgraphs, we perform a simple statistical similarity

Table 4: Structural attributes comparison of the 12 most frequently occurring subgraphs in a dataset of 750 graphs.

Subgraph ID	Nodes	Edges	Radius	Diameter	Density	Frequency
Subgraph #1	9	10	1	2	0.1388888	54
Subgraph #2	10	9	1	2	0.1	53
Subgraph #3	9	8	2	4	0.1111111	49
Subgraph #4	7	12	2	2	0.2857142	38
Subgraph #5	14	17	3	5	0.0934065	32
Subgraph #6	9	10	3	4	0.1388888	28
Subgraph #7	10	9	2	4	0.1	27
Subgraph #8	13	16	3	4	0.1025641	21
Subgraph #9	17	22	3	5	0.0808823	20
Subgraph #10	6	7	2	3	0.2333333	15
Subgraph #11	16	20	3	5	0.0833333	13
Subgraph #12	11	10	4	8	0.0909091	12

analysis using the subgraphs' structural properties to find the most frequently appearing subgraphs in the dataset. In the analysis, we obtain graphs' radius, diameter and density attributes, as well as a list of their node and edge labels, and match graphs that agree in all criteria. Properties and visualizations of the 12 subgraphs with the highest appearance frequency in our dataset of 750 generated graphs are listed in Table 4 and visualized in Figure 17. By examining the data in Table 4, we note that the most frequently occurring subgraphs are fairly small in size (under 11 nodes and edges), and do not exhibit any complex connections between their nodes. This result is to be expected since by virtue of simplicity, one would expect such subgraphs to occur fairly frequently and not be particularly indicative of a deliberate design pattern. However, we also note that the algorithm is also able to identify a number of somewhat complex subgraphs, such as subgraphs

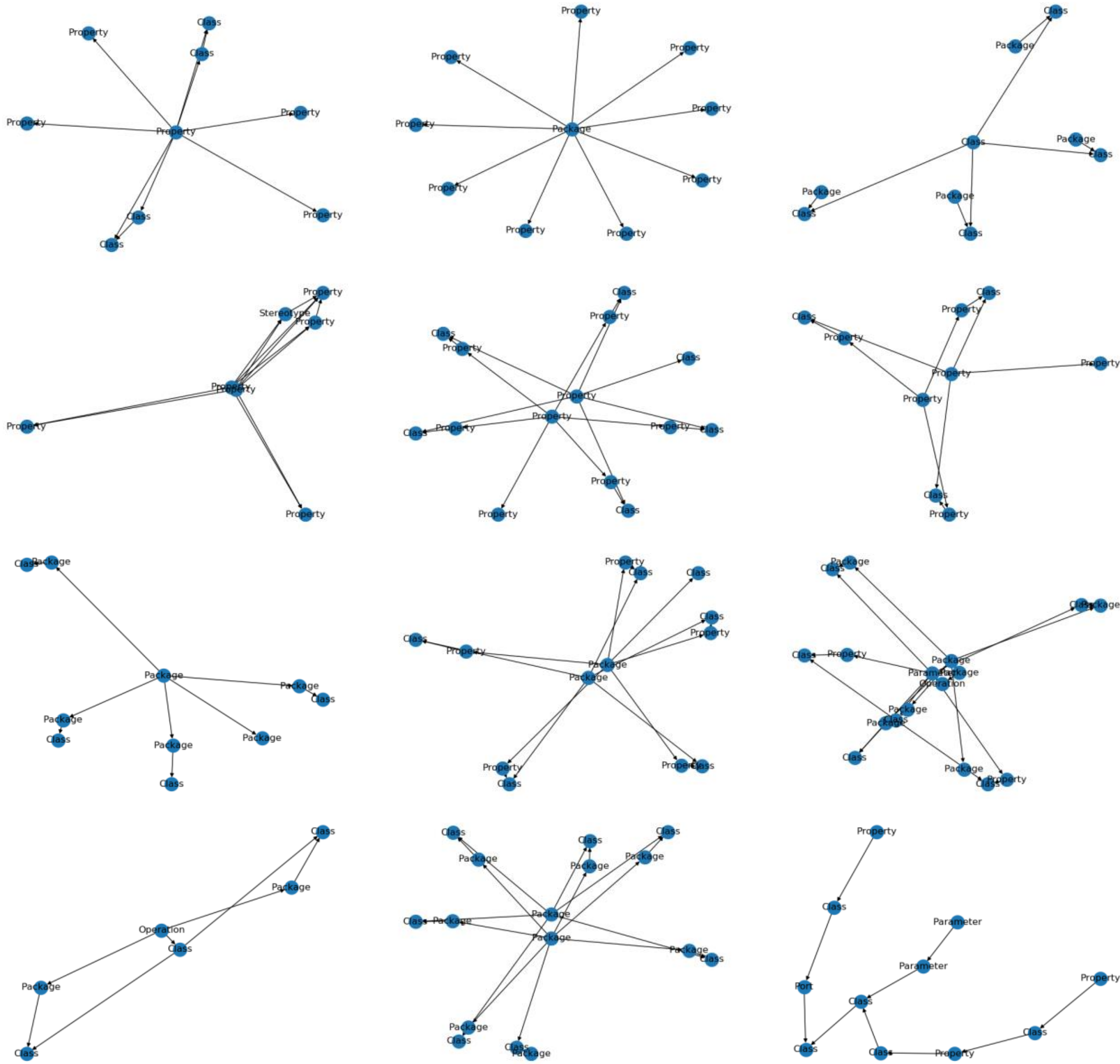

Figure 17: Visualization of the 12 most frequently occurring subgraphs in our dataset of 750 generated graphs.

number five, nine, and eleven, that may have some functional significance in operational systems. The extracted subgraphs could be of help to system designers and network architects who may be interested in identifying some commonly-used system components in their area of interest.

These results highlight the fact that through unsupervised learning and with enough realistic architecture graphs, one may be able to extract unknown design patterns in physical and

mechanical systems. Such a feat has historically been reserved for software systems, and only for a few well-defined patterns. Our framework is entirely general and applicable to a wide variety of system architectures, as it makes no assumptions about network structure or data distribution. Future work in this area could explore different embedding and clustering techniques, as well as ways to infer functionality of extracted subgraph patterns.

CHAPTER 5: CONCLUSIONS

This thesis examined the use of network optimization and machine learning methods to solve two distinct network-centric problems relating to operational systems. The first problem, discussed in Chapter 3, involved assessing the resilience of the US air transportation network against worst-case disruptions. To this end, we developed a tri-level optimization program, and used passenger origin and destination data provided by the Bureau of Transportation Statistics to model the network's response to various disruption scenarios. We also investigated how the network can recover optimally from such disruptions, as well as the possible infrastructure planning steps that could be taken to augment the resilience of the air network. Our results suggest that establishing alternative high-speed transportation options between the two coasts of the United States could potentially be the investment option with the highest potential positive impact on the air network's resilience. The thesis' second problem of interest, detailed in Chapter 4, was centered on the unsupervised discovery of design patterns in system architecture graphs. Our developed framework was designed to accept one or more such architecture graphs and, through a series of mathematical operations and machine learning models, to output the design patterns that have the highest potential to occur most in graphs with the same distribution as that of the input graphs. This goal is accomplished through the unsupervised clustering of the graphs, guided by their latent representations and adjacency matrices using the agglomerative clustering algorithm. Our results indicate that finding and studying patterns in operational non-software systems, architectures that historically have not been considered in pattern analyses, could not only be possible but also potentially beneficial in the design and optimization processes of these systems. Future work directions could include applying the framework to more architecture data, and a more rigorous, subject-matter specific analysis of the potential significance of certain discovered patterns.

REFERENCES

[1] B. Bowler and S. Parminter, "Network," *Network*. Oxford University Press, Oxford, 1996.

[2] J. McAuley and J. Leskovec, "Learning to Discover Social Circles in Ego Networks," *Advances in Neural Information Processing Systems,* pp. 539-547, 2012.

[3] L. Euler, "Solutio problematis ad geometriam situs pertinentis," *Comment. Acad. Sci. U. Petrop 8*, pp. 128–40, 1736.

[4] D. P. Bertsekas, *Network optimization: continuous and discrete models*. Belmont, MA: Athenea Scientific, 1998.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] R. Bellman, "On a routing problem," Quarterly of Applied Mathematics, vol. 16, no. 1, pp. 87–90, 1958.

[7] L. R. Ford and D. R. Fulkerson, "Maximal Flow Through a Network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[8] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Doklady*, vol. 11, pp. 1277-1280, 1970.

[9] J. B. Orlin, "A polynomial time primal network simplex algorithm for minimum cost flows," *Mathematical Programming*, vol. 78, no. 2, pp. 109–129, 1997.

[10] Z. Zhu, J. Tang, S. Lambotharan, W. H. Chin, and Z. Fan, "An integer linear programming based optimization for home demand-side management in smart grid," *2012 IEEE PES Innovative Smart Grid Technologies (ISGT)*, 2012.

[11] A. Richards and J. How, “Aircraft trajectory planning with collision avoidance using mixed integer linear programming,” *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, 2002.

[12] M. Carrion and J. Arroyo, “A Computationally Efficient Mixed-Integer Linear Formulation for the Thermal Unit Commitment Problem,” *IEEE Transactions on Power Systems*, vol. 21, no. 3, pp. 1371–1378, 2006.

[13] D. Solow, “Linear and Nonlinear Programming,” *Wiley Encyclopedia of Computer Science and Engineering*, 2007.

[14] R. Hemmecke, M. Köppe, J. Lee, and R. Weismantel, “Nonlinear Integer Programming,” *50 Years of Integer Programming 1958-2008*, pp. 561–618, 2009.

[15] M. Geoffrion, “Generalized benders decomposition,” *Journal of Optimization Theory and Applications*, vol. 10, no. 4, pp. 237–260, 1972.

[16] K. Gupta and V. Ravindran, “Branch and bound experiments in convex nonlinear integer programming,” *Management Science*, vol. 31, no. 12, pp. 1533–1546, 1985.

[17] M. A. Duran and I. E. Grossmann, “An outer-approximation algorithm for a class of mixed-integer nonlinear programs,” *Math Programming*, vol. 36, p. 307, 1986.

[18] C. Bragalli, C. D’Ambrosio, J. Lee, A. Lodi, and P. Toth, “An MINLP Solution Method for a Water Network Problem,” *Lecture Notes in Computer Science Algorithms – ESA 2006*, pp. 696–707, 2006.

[19] A. Martin, M. Möller, and S. Moritz, “Mixed Integer Models for the Stationary Case of Gas Network Optimization,” *Mathematical Programming*, vol. 105, no. 2-3, pp. 563–582, 2005.

[20] W. Murray and U. V. Shanbhag, “A Local Relaxation Approach for the Siting of Electrical Substations,” *Computational Optimization and Applications*, 2006.

[21] President’s Commission on Critical Infrastructure Protection, *Critical Foundations: Protecting America’s Infrastructure*, October 1997.

[22] G. Brown, M. Carlyle, J. Salmerón, and K. Wood, “Defending Critical Infrastructure,” *Interfaces*, vol. 36, no. 6, pp. 530–544, 2006.

[23] H. von. Stackelberg, R. Hill, D. Bazin, and L. Urch, *Market structure and equilibrium*. Heidelberg: Springer, 2011.

[24] J. Salmeron, K. Wood, and R. Baldick, “Worst-Case Interdiction Analysis of Large-Scale Electric Power Grids,” *IEEE Transactions on Power Systems*, vol. 24, no. 1, pp. 96–104, 2009.

[25] J. Pita, M. Jain, J. Marecki, F. Ordőñez, C. Portway, M. Tambe, C. Western, P. Paruchuri, S. Kraus, and M. Tambe, “Deployed ARMOR Protection: The Application of a Game-Theoretic Model for Security at the Los Angeles International Airport,” *Security and Game Theory*, pp. 67–87.

[26] N. T. Boardman, B. J. Lunday, and M. J. Robbins, “Heterogeneous surface-to-air missile defense battery location: a game theoretic approach,” *Journal of Heuristics*, vol. 23, no. 6, pp. 417–447, 2017.

[27] D. L. Alderson, G. G. Brown, W. M. Carlyle, and R. K. Wood, “Assessing and Improving the Operational Resilience of a Large Highway Infrastructure System to Worst-Case Losses,” *Transportation Science*, 2017.

[28] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: from theory to algorithms*. Cambridge: Cambridge University Press, 2017.

[29] D. Ciresan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[30] A. Sallab, M. Abdou, E. Perot, and S. Yogamani, “Deep Reinforcement Learning framework for Autonomous Driving,” *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[31] H. Sethy, A. Patel, and V. Padmanabhan, “Real Time Strategy Games: A Reinforcement Learning Approach,” *Procedia Computer Science*, vol. 54, pp. 257–264, 2015.

[32] Z. Zhang, P. Cui, and W. Zhu, “Deep learning on graphs: A survey,” *arXiv preprint arXiv:1812.04202*, 2018.

[33] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[34] P. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, “Interaction networks for learning about objects, relations and physics,” *Advances in Neural Information Processing Systems*, 2016.

[35] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907,* 2016.

[36] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 14*, 2014.

[37] J. Weston, F. Ratle, and R. Collobert, “Deep learning via semi-supervised embedding,” *Proceedings of the 25th international conference on Machine learning - ICML 08*, 2008.

[38] T. Kipf and M. Welling, " Variational graph auto-encoders," *arXiv preprint arXiv: 1611.07308,* 2016.

[39] I. Goodfellow, J. Pouget-Abadie, M. Mehdi, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, pp. 2672-2680, 2014.

[40] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *arXiv preprint arXiv:1511.06434*, 2015.

[41] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4401-4410, 2019

[42] K. Cho, B. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation,” *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[43] A. Graves, A. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[44] Y. Ma, Z. Guo, Z. Ren, E. Zhao, J. Tang, and D. Yin, "Dynamic graph neural networks," *arXiv preprint arXiv:1810.10627*, 2018.

[45] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," *arXiv preprint arXiv:1802.08773,* 2018.

[46] (2017). *Bureau of Transportation Statistics, Airline Origin and Destination Survey*. [Online]. Available: https://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=125.

[47] K. H. Thompson and H. T. Tran, “Operational Perspectives Into the Resilience of the U.S. Air Transportation Network Against Intelligent Attacks,” *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–11, 2019.

[48] F. Margot, M. Queyranne, and Y. Wang, “Decompositions, Network Flows, and a Precedence Constrained Single-Machine Scheduling Problem,” *Operations Research*, vol. 51, no. 6, pp. 981–992, 2003.

[49] E. Israeli and R. K. Wood, “Shortest-path network interdiction,” *Networks*, vol. 40, no. 2, pp. 97–111, 2002.

[50] N. Alguacil, A. Delgadillo, and J. M. Arroyo, “A trilevel programming approach for electric grid defense planning,” *Computers & Operations Research*, vol. 41, pp. 282–290, 2014.

[51] D. L. Alderson, G. G. Brown, W. M. Carlyle, and R. K. Wood, “Solving Defender-Attacker-Defender Models for Infrastructure Defense,” *12th INFORMS Computing Society Conference*, Jan. 2011.

[52] (2018). *Air Traffic By The Numbers, Federal Aviation Administration*. [Online]. Available: https://www.faa.gov/air_traffic/by_the_numbers

[53] *U.S. International Air Passenger and Freight Statistics*. US Department of Transportation, Washington, DC, USA, 2017.

[54] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. New Dehli: Pearson Education, 2015.

[55] J. Dong, Y. Zhao, and T. Peng, “Architecture and Design Pattern Discovery Techniques-A Review,” *Software Engineering Research and Practice*, pp. 621-627, 2007.

[56] C. Kramer and L. Prechelt, “Design recovery by automated search for structural design patterns in object-oriented software,” *Proceedings of WCRE 96: 4rd Working Conference on Reverse Engineering*, 1996.

[57] R. Keller, R. Schauer, S. Robitaille, and P. Page, “Pattern-based reverse-engineering of design components,” *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, 1999.

[58] A. Blewitt, A. Bundy, and I. Stark, “Automatic verification of Java design patterns,” *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001.

[59] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, “Design pattern mining enhanced by machine learning,” *21st IEEE International Conference on Software Maintenance (ICSM05)*, 2005.

[60] G. Y-G, H. Sahraoui, and F. Zaidi, “Fingerprinting design patterns,” *11th Working Conference on Reverse Engineering*, 2004.

[61] Z. Zhang, Q. Li, and K. Ben, “A new method for design pattern mining.” *Proceedings of the 3rd International Conference on Machine Learning and Cybernetics*, 2004.

[62] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, “Design Pattern Detection Using Similarity Scoring,” *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[63] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, “Automatic design pattern detection,” *MHS2003. Proceedings of 2003 International Symposium on Micro-mechatronics and Human Science (IEEE Cat. No.03TH8717)*, 2003.

[64] J. Niere, W. Schafer, J. Wadsack, L. Wendehals, and J. Welsh, “Towards pattern-based design recovery,” *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, 2002.

[65] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, no. 1, pp. 47–97, 2002.

[66] E. Airoldi, M. Edoardo, D. Blei, S. Fienberg, and E. Xing, "Mixed membership stochastic blockmodels," *Journal of machine learning research*, pp. 1981-2014, 2008.

[67] D. J. Watts and S. H. Strogatz, “Collective dynamics of small-world networks,” *The Structure and Dynamics of Networks*, 2011.

[68] G. Robins, P. Pattison, Y. Kalish, and D. Lusher, “An introduction to exponential random graph (p*) models for social networks,” *Social Networks*, vol. 29, no. 2, pp. 173–191, 2007.