DISSERTATION

INTEGRATING GEOMETRIC DEEP LEARNING WITH A SET-BASED DESIGN APPROACH FOR THE EXPLORATION OF GRAPH-BASED ENGINEERING SYSTEMS

Submitted by

Anthony Sirico Jr.

Department of Systems Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2024

Doctoral Committee:

Advisor: Daniel R. Herber

Haonan Chen
Steven Simske
Steven Conrad

ABSTRACT

INTEGRATING GEOMETRIC DEEP LEARNING WITH A SET-BASED DESIGN APPROACH FOR THE EXPLORATION OF GRAPH-BASED ENGINEERING SYSTEMS

Many complex engineering systems can be represented in a topological form, such as graphs. This dissertation introduces a framework of Graph-Set-Based Design (GSBD) that integrates graph-based techniques with Geometric Deep Learning (GDL) within a Set-Based Design (SBD) approach to address graph-centric design problems. We also introduce Iterative Classification (IC), a method for narrowing down large datasets to a subset of more promising and feasible solutions. When we combine the two, we have IC-GSBD, a methodological framework where the primary goal is to effectively and efficiently seek the best-performing solutions with lower computational costs.

IC-GSBD is a method that employs an iterative approach to efficiently narrow down a graph-based dataset containing diverse design solutions to identify the most useful options. This approach is particularly valuable as the dataset would be computationally expensive to process using other conventional methods. The implementation involves analyzing a small subset of the dataset to train a machine-learning model. This model is then utilized to predict the remaining dataset iteratively, progressively refining the top solutions with each iteration. In this work, we present two case studies demonstrating this method.

In the first case study utilizing IC-GSBD, the goal is the analysis of analog electrical circuits, aiming to match a specific frequency response within a particular range. Previous studies generated 43,249 unique undirected graphs representing valid potential circuits through enumeration techniques. However, determining the sizing and performance of these circuits proved computationally expensive. By using a fraction of the circuit graphs and their performance as input data for a classification-focused GDL model, we can predict the performance of the remaining graphs with

favorable accuracy. The results show that incorporating additional graph-based features enhances model performance, achieving a classification accuracy of 80% using only 10% of the graphs and further subdividing the graphs into targeted groups with medians significantly closer to the best and containing 88.2 of the top 100 best-performing graphs on average using 25% of the graphs.

In the second case study, we demonstrate the IC-GSBD methodological framework on aircraft thermal management systems (TMSs). Utilizing another previously created dataset through enumeration, the approach employs directed graphs representing TMSs and GDL to predict critical characteristics such as compilability and simulatability. The findings indicate that incorporating graph-based features using Principal Component Analysis (PCA) significantly enhances GDL model performance, achieving an accuracy of 98% for determining a graph's compilability and simulatability while using only 5% of the data for training. Additionally, iterative classification methods successfully segment the total set of graphs into specific groups, including an average of 71.3 of the top 100 highest-performing graphs, by training on 30% of the data.

This work demonstrates that IC-GSBD can significantly enhance the process of complex systems design by leveraging advanced computational techniques, reducing computational overhead, and offering a scalable model for applying machine learning to certain engineering design problems with many options that are computationally expensive to evaluate.

ACKNOWLEDGEMENTS

Working with Dr. Daniel R. Herber on this endeavor for the past four years has been an honor. His knowledge and guidance have given me the tools necessary to complete this and prepared me for what lies ahead. I would also like to thank my dissertation committee for their time and feedback.

I would like to thank my parents, Tony and Donna, and my brother Jason for their love and continued support throughout this process.

Most importantly, I would like to thank my wife, Tori, and son, Anthony, because none of this would have been possible without them. It was no easy task to tolerate the conference calls and late-night work, but I am thankful every day for having them in my life. Their love and support throughout this endeavor is what really made this all possible.

DEDICATION

To my beautiful wife, Tori, and my son, "Little" Anthony

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

GDL Geometric Deep Learning *on page(s):* 9, 26, 56, 60, 78, 86

GNN Graph Neural Network *on page(s):* 9, 28

GSBD Graph-Set-Based Design *on page(s):* 3, 8, 56

IC Iterative Classification *on page(s):* 9, 52

IC-GSBD Iterative Classification for Graph-Set-Based Design *on page(s):* 60, 70, 72, 86

MCC Matthews Correlation Coefficient *on page(s):* 51, 79

PCA Principal Component Analysis *on page(s):* 39, 81

SBD Set-Based Design *on page(s):* 7, 14, 56, 60, 86

LIST OF SYMBOLS

E A set of Edges *on page(s):* 17

F_N False Negatives, extracted from the confusion matrix *on page(s):* 48

F_P False Positives, extracted from the confusion matrix *on page(s):* 48

G A single graph *on page(s):* 17

$J(G_i)$ Performance value of graph G_i *on page(s):* 35

N_{all} The size of the set $\mathcal{G}_{all}$ *on page(s):* 53

N_{known} The number of graphs in the "known" dataset *on page(s):* 53

$N_{unknown}$ The number of graphs in the "unknown" dataset *on page(s):* 53

T_N True Negatives, extracted from the confusion matrix *on page(s):* 48

T_P True Positives, extracted from the confusion matrix *on page(s):* 48

V A set of vertices *on page(s):* 17

$\bar{J}$ The median performance value of set of graphs $\mathcal{G}$ *on page(s):* 37

$\mathbf{B}$ An incidence matrix *on page(s):* 18

$\mathbf{L}(G_i)$ An array of graph-level labels for graph G_i *on page(s):* 37

$\mathbf{W}$ A weighted edge matrix *on page(s):* 20

$\mathbf{X}$ Feature set *on page(s):* 20

$\mathcal{G}_{all}$ An entire set of graphs that will eventually be split into "known" and "unknown" sets *on page(s):* 53

$\mathcal{G}_{known}$ The set of graphs whose graph labels are *known* and is used to train the model *on page(s):* 53

$\mathcal{G}_{training}$ The set of graphs that the model is intended to train on *on page(s):* 53

$\mathcal{G}_{unknown}$ The set of graphs whose graph labels are *unknown* and the model is used to predict *on page(s):* 53

$\mathcal{G}_{validation}$ The set of graphs that the model will test or *validate* itself during the training process *on page(s):* 53

$\mathcal{G}$ A set of graphs *on page(s):* 35, 53

l_c A graph label from the graph label array $\mathbf{L}(G_i)$ *on page(s):* 37

m A Geometric Deep Learning model *on page(s):* 55

n The number of iterations to be completed during iterative classification *on page(s):* 55

$|\mathcal{G}_{known}|_{min}$ The minimum size of $\mathcal{G}_{known}$ in order to generate an effective prediction model *on page(s):* 55

Chapter 1

Introduction

> "The main task of engineers is to apply their scientific and engineering knowledge to the solution of technical problems, and then to optimize those solutions within the requirements and constraints set by material, technological, economic, legal, environmental and human-related considerations."
>
> — *G. Pahl, et al.* [1]

The need for innovative solutions has never been higher. As highlighted by Pahl et al., the essence of engineering design lies in optimizing solutions within a myriad of constraints [1]. This quote encapsulates the multi-dimensional challenges faced by engineers, emphasizing the necessity for innovative methodologies that can effectively navigate and optimize complex system designs. In this chapter, we introduce and justify a new type of framework for the engineering design process, a framework that includes Machine Learning (ML). We begin by discussing the contextual background and motivation behind this work, followed by a discussion of the role of Artificial Intelligence (AI)/Machine Learning (ML) in systems engineering. We will walk through the evolution of the engineering design process and the pressing need for innovative solutions. Lastly, an introduction to the new framework is given.

1.1 Contextual Background and the Need for Innovation

Recent advancements in engineering design methodologies have increasingly emphasized the need for more comprehensive and flexible approaches to navigate the complexities of system design and optimization. *Complexity*, in this context, refers to the degree of interconnections and the level of difficulty to solve or analyze the system [2]. Increasing system complexity continues to be one of the biggest challenges facing engineering design and manufacturing today, as these systems operate in an environment of increasing change and uncertainty [3]. As we venture deeper into

the 21st century, the engineering domain confronts many challenges posed by the intricacy and multifaceted nature of modern systems.

Engineering design sits at the heart of innovation by integrating creative ideas with practical applications. Designers account for not only the technical aspect of the system but the economic properties and commercial importance of timely and efficient system development, but it is also important to have well-defined design methods that find solutions [1]. That said, designers have to manage increasing complexity, which necessitates the call for advancements in the overall design process.

The initial stages of the design process, known as *conceptual design*, involve defining the design space, which can lead to a combinatorial explosion due to the vast array of design parameters and constraints. This challenge makes identifying optimal solutions a daunting task, requiring strategies that can efficiently explore a multitude of possibilities. With an increase in the number of possible solutions comes an increase in the computational cost. Detailed simulations and optimization of potential designs incur a significant computational cost, especially during the early stages of design, when numerous iterations may be necessary to converge on a viable solution. Modern engineering systems often span multiple domains, making it necessary to navigate interdisciplinary constraints and balance competing requirements to achieve optimal system performance.

The escalating complexity of engineering tasks and the exponential growth in design possibilities demand solutions that mitigate these challenges and discover new opportunities for innovation and efficiency. As we continue to advance systems in the years to come, these challenges will not only continue to increase but hinder progress toward a more technologically prosperous future where we can focus more on systems' intricate and conscientious aspects. In this work, we seek to explore the need for an innovative solution that addresses the challenges that come with the increasing complexity of systems.

As methodologies continue to improve over the years, one thing is evident: each method builds upon the last, which is no different here. We need methodologies that continue to optimize the design process and foster innovation by enabling the exploration of previously infeasible design

spaces. Innovative design methodologies should be able to streamline the exploration of design spaces by offering tools and techniques for rapidly identifying and focusing on viable design options from the outset, minimizing time and resource expenditure.

With the introduction of machine learning and artificial intelligence, new methodologies should be able to minimize computational demands by employing predictive models to estimate system performance and feasibility, reducing reliance on computationally intensive simulations for every potential design iteration. Lastly, new methods should be able to integrate multidisciplinary design constraints by providing a framework for incorporating and balancing diverse design constraints, facilitating the development of solutions that address multifaceted system requirements.

These needs call for a significant change in the engineering design process. Traditional foundational methodologies are becoming increasingly inadequate when applied to the more multifaceted and dynamic engineering landscape, and the need for new methodologies capable of handling these solutions is more pressing than ever. This dissertation introduces a machine learning method based on Set-Based Design techniques called Graph-Set-Based Design (GSBD), aiming to address these challenges by utilizing data sets represented as graphs and machine learning techniques to narrow down possible solutions based on specific constraints automatically. In the following section, we provide an overview of the necessity for Artificial Intelligence and Machine Learning (AI/ML) techniques from two perspectives: a strategic standpoint and an engineering standpoint. This dual perspective gives insight into the critical role of AI/ML in enhancing decision-making processes and maintaining technological superiority.

1.2 Systems Engineering with Artificial Intelligence (AI) and Machine Learning (ML)

Artificial Intelligence (AI) is a pivotal and sensitive subject, whether discussing its role in everyday life or within the engineering design process. There is an increasing demand for faster decision-making capabilities than what humans can currently provide [4, Chapter 1, pp. 3–4]. China's military ambitions exemplify this demand, including the "intelligentization" of its forces

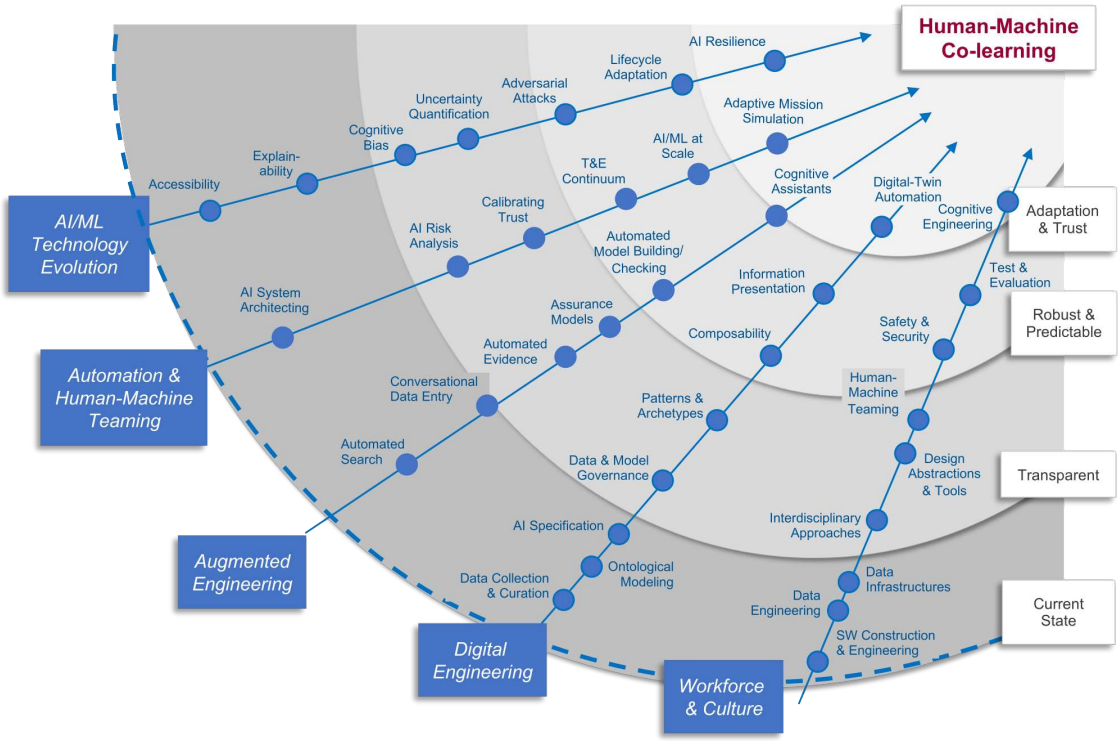


Figure 1.1: The AI4SE and SE4AI roadmap developed by the Systems Engineering Research Center (SERC) [9]

by integrating AI and other emerging technologies into its joint force, which is believed to give Beijing a competitive advantage over Washington [5]. Advances in AI-enabled weapons by China could upset the military balance, threatening global security and strategic stability [6]. The U.S. President has described the relationship between Washington and Beijing as descending into "extreme" strategic competition across diplomatic, economic, military, and governance arenas [7]. These developments underscore the critical need for integrating AI/ML within Systems Engineering to maintain strategic and technological superiority. Over the past decade, the U.S. military began increasing the presence of AI within its systems. In 2019, the Defense Innovation Board (DIB) advised the Department of Defense (DoD) to ensure that any AI used is "responsible, equitable, traceable, reliable, and governable" [8]. This further solidifies the necessity of keeping a human in the loop.

Figure 1.1 shows the current roadmap for AI and Autonomy from the Systems Engineering Research Center (SERC), whose goal is to lead the transformation of systems engineering to dy-

namic processes that leverage the speed and rigor of rapidly evolving modeling, simulation, and analysis. The roadmap is entitled, "AI4SE and SE4AI", two terms that may seem related but are also very different. AI4SE, or Artificial Intelligence for Systems Engineering, is the application of augmented intelligence and ML techniques to support SE practice. For example, in this study, ML techniques are being used to support a particular SE process, the down-selection of a design based on a set of criteria. While SE4AI is defined as the application of SE methods to the design and operation of learning-based systems [4, Chapter 3, pp. 48-49]. According to McDermott et al. in Chapter 3 of Ref. [4], entitled "Artificial Intelligence and Future of Systems Engineering", these terms have become metaphors for an upcoming rapid revolutionary phase in the SE community.

The vector, Automation & Human-Machine Teaming, outlines a clear progression toward Human-Machine Co-Learning, starting with AI System Architecting. McDermott et al. note that this progression will evolve as automation is integrated into increasingly complex systems. System architectures must support learning, adaptation, and more agile change processes to accommodate this evolution [4, Chapter 3, pp. 56]. That is why, in this study, we present a new methodology capable of utilizing AI/ML techniques and adapting to new problems that vary in complexity, as shown in the two case studies.

This look into AI/ML within the U.S. military and SE processes highlights the crucial need to integrate AI and ML within systems engineering to tackle the rising complexity in design and decision-making. We need to examine the evolution of the design process over the years to better understand our proposed methodological framework.

1.2.1 Evolution of Engineering Design Frameworks & Tools

It was not until the early twentieth century that modern approaches to design were conceived. Systematic Design was developed, accepted, and used for decades, turning customer requirements into working products. This method consists of four phases: product planning, conceptual design, embodiment design, and detail design. It is a structured approach to solving design problems. First, the process is broken down into manageable steps, including requirements analysis, exploration of

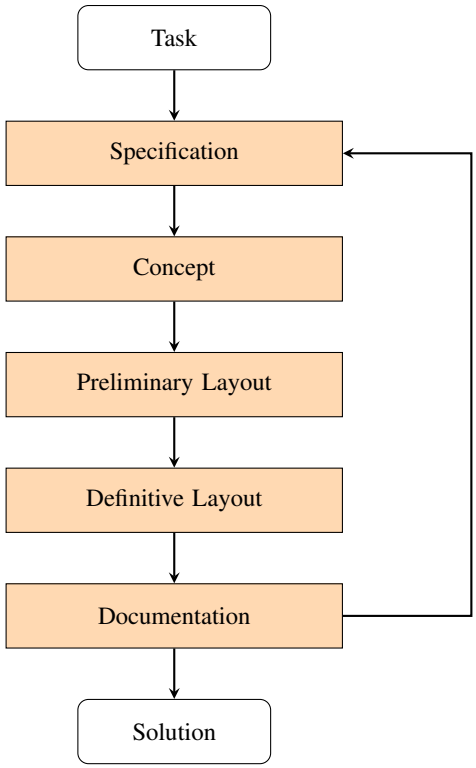


Figure 1.2: The Systematic Design workflow

potential solutions, design development, and testing and refinement. This method ensures that all aspects of the design problem are considered and addressed, which leads to more reliable solutions [1].

The Systematic Design approach has several advantages but also faces challenges. As shown in Figure 1.2, the process is very linear, which makes it difficult to adapt to unforeseen changes or new information during the design phase. Due to its structured approach, strictly following its steps can stifle creativity and innovation. Additionally, in the conceptual design phase, there is limited room for analyzing large datasets of multiple feasible solutions. According to Pahl et al., Systematic Design requires careful planning of activities, timing, scheduling, project and product costs, which can make the process overly complex, leading to increased project overhead and delays [1].

Then came Computer-Aided Technologies (CATs), which aided in designing, planning and scheduling, analyzing, manufacturing, and assembling products and production systems. CAT revolutionized the design process by integrating computational tools, which enhanced precision

and efficiency. Developed for manufacturing by Patrick Hanratty in 1957, innovations such as Sketchpad (1960), where a user can generate (x, y) plots on a Graphical User Interface GUI, led to Computer-Aided Design and Drawing (CADD) developed by McDonell-Douglas in 1966. In 1982, AutoCAD was developed by Autodesk as the first CAD software for personal computers (PCs), and then in 1994, version 13 was released with the capability of three-dimensional (3D) modeling [10, 11].

In the 1980s, Knowledge-Based Engineering (KBE) was developed. KBE uses product and process knowledge captured and stored in dedicated software applications to enable its direct exploitation and reuse in the design of new products and variants. It also enables geometric modeling and manipulation, differing from CAD in that it encapsulates the knowledge and rationale behind ideas rather than just the geometry, as CAD does [12].

Similar to Systematic Design, KBE also has its drawbacks. One drawback is that building a database for KBE is very time-consuming. It takes time to train a KBE system to perform basic tasks. Another downside is that the processes the KBE system goes through to generate its results are unknown, which can give a false sense of security and lead to problems. That's why it's crucial to understand the processes and transfer the knowledge back to the engineering team [13].

Set-Based Design (SBD), sometimes referred to as Set-Based Concurrent Engineering (SBCE), was developed in the 1990s. It explores multiple design solutions simultaneously and progressively by narrowing down the options based on meeting specific constraints. It enables designers to focus on current development by avoiding convergence on a single solution [14, 15]. We describe SBD in more detail in Chapter 2.1.

In the 2000s, Model-Based Systems Engineering (MBSE) was developed. MBSE shifts the focus from document-based to model-centric processes, supporting the full lifecycle of system development. MBSE also provides improved management of system complexity and enhances communication among stakeholders [16].

The evolution of engineering design methodologies demonstrates a progression from structured, systematic approaches to more advanced technology and data-driven methods. Initially,

Systematic Design laid the groundwork with its structured problem-solving techniques. The introduction of Computer-Aided Technologies (CATs) and Computer-Aided Design and Drafting (CADD) tools improved knowledge management and design efficiency for certain kinds of problems. This evolution continued with Knowledge-Based Engineering (KBE), which further enhanced the design and reuse of knowledge. Set-Based Design (SBD) and Model-Based Systems Engineering (MBSE) followed, integrating these tools and methodologies to streamline and optimize the design process. This progression demonstrates the increasing complexity and efficiency in design processes, setting the stage for the incorporation of Artificial Intelligence and Machine Learning (AI/ML) as a next step in advancing engineering design.

In the next section, we introduce our proposed methodological framework, leveraging these advancements to further enhance the design process through the integration of machine learning and graph-based techniques.

1.3 Graph-Set-Based Design (GSBD) Introduction

The Graph-Set-Based Design (GSBD) framework is an enhancement to current engineering design processes. This framework integrates graph-based representations of systems, a Set-Based Design approach, and Geometric Deep Learning (GDL) to provide a novel approach to system design and optimization.

Mathematical graphs effectively represent many systems and decisions because they capture discrete compositional and relational information. For decades, various studies have used different graph representations to address specific problems [17–26]. For well over a century, researchers have utilized graph enumeration to understand engineering design problems and aid in decision-making [18, 19, 27–29]. Today, many engineering design problems, including the construction of the "system architecture", have grown in scope and complexity to a level where traditional discrete and continuous representations are insufficient to adequately represent the system [18]. The system architecture is a conceptual model that captures some or all the structure, behavior, rules, and other aspects of a product, process, or element [30]. It often serves as the foundation for the product is

design, construction, and operation. Many studies have focused on effectively representing system architectures using graph theory [26, 31, 32], aiming to generate a set of useful architectures that are feasible within given constraints. Additionally, one or more value metrics (e.g., performance or cost) may be determined for the alternative architectures to help designers evaluate and sort through the candidates.

Graph enumeration produces a complete and ordered listing of potential graphs for a prescribed structure [27, 33, 34]. However, as mentioned in the previous section, depending on the problem and the chosen representation, this approach can result in a vast number of potential solutions. With that comes an exponential rise in computational costs, making the decision-making process for designers or system architects increasingly challenging.

This dissertation also introduces Iterative Classification (IC) in response to the need for a method for determining more promising and effective solutions, when combined with GSBD, becoming IC-GSBD, offering a novel approach to system design and optimization.

As a major final component, deep learning refers to machine learning models that consist of multiple processing layers capable of learning data representations at various levels of abstraction [35]. The term "deep" indicates the presence of a larger number of hidden layers within the neural network. GDL is a broad term that encompasses an emerging technique that generalizes neural networks to both Euclidean and non-Euclidean domains, including graphs, manifolds, meshes, and string representations [36], utilizing a *Graph Neural Network (GNN)*. Essentially, GDL includes approaches that incorporate information about the structure space and symmetry properties of input variables to enhance the quality of the data captured by the model. GDL holds significant potential and is widely applied in various scientific fields such as molecular representations [37–39], architecture [40], and the medical field [41, 42].

In engineering design, GNNs have been applied to a range of tasks, including airfoil design [43], structural mechanics with mesh-based physics simulations [44, 45], wind-farm power estimation [46], distributed circuit design [47], and decision-making processes in shared mobility systems [48]. Additionally, GNNs have facilitated component function classification by training

on assembly and flow relationships [49]. However, none of these references specifically address the common engineering task of down-selecting from a finite set of potential options.

Here, we are particularly motivated by graph-centric design problems where generating many potential graphs is feasible, but determining the value or performance of each option is too expensive. Previous work has shown promising results on the dataset used in one of the case studies in this work but required a significant training set size and time [50]. The proposed approach uses GDL to reduce the computational expense of classifying what might be "good" or "bad" options. It utilizes much smaller training sets, with a trade-off in classification accuracy.

The IC-GSBD framework requires models of complex systems as graphs by capturing the relationships and dependencies between components and offering a natural and intuitive means of understanding system architecture. IC-GSBD can leverage GDL for performance evaluation by using it to predict the performance of graph-represented systems. This approach facilitates the early identification of promising design directions without the need for many detailed simulations. Lastly, IC-GSBD iteratively sifts large datasets, aiding in the down-select process to efficiently narrow the focus to more promising design alternatives, giving designers the ability to analyze the results more intimately. Unlike traditional methods that may require manipulating or refining designs iteratively at a granular level, IC-GSBD focuses on leveraging predictive modeling to inform decision-making. Figure 1.3 shows the basic concept of IC-GSBD, where we begin with a set of possible solutions after the graph enumeration step has been accomplished. Through prioritizing exploration and intelligent selection from vast design spaces without directly altering the graph-based representations of systems, we down-select to a subset of the more useful systems.

The capability of IC combined with GSBD ensures a more informed selection process by dynamically adapting to new insights and data, effectively reducing the iterative cycles typically required in other approaches such as Set-Based Design, discussed further in Chapter 2. This capability is crucial in early design stages, where the ability to rapidly gauge the impact of design decisions can significantly streamline the design process, reduce computational overhead, and guide the focus towards the most promising design avenues. Moreover, GSBD embodies the SBD philos-

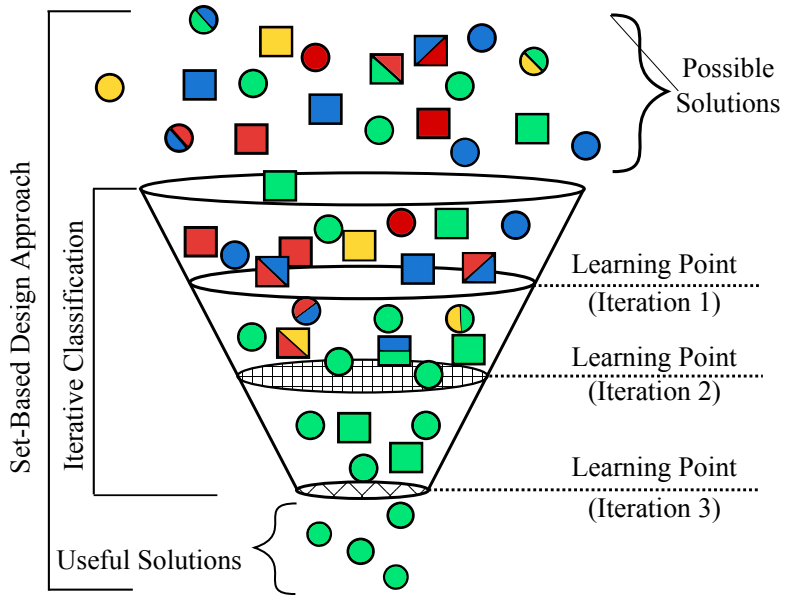


Figure 1.3: The basic concept of Graph-Set-Based Design and Iterative Classification

ophy by enabling a systematic and iterative design space exploration, yet it transcends traditional limitations by leveraging GDL's analytical prowess. This advancement is achieved by utilizing advanced machine learning to interpret and analyze the intricate relationships within design parameters. Thus, the depth of analysis is significantly elevated beyond what is conventionally possible with SBD alone. In doing so, IC-GSBD manages complexity more effectively and enhances the efficiency and depth of the design exploration process. IC-GSBD stands as a testament to the potential of combining advanced machine learning techniques with established design methodologies to navigate the challenges inherent in designing complex engineering systems.

1.4 Dissertation Objectives and Structure

This introduction has provided an overview of the current challenges and possible solutions within the landscape of engineering design. There are three primary goals that we intend to accomplish within this work:

1. Provide support in the down-selection process by reducing the number of candidate graphs (e.g., from thousands to less than a hundred of the best potential options)

2. Significantly reduce the computational cost for problems requiring complex evaluations of graphs, providing feasible solutions that meet stakeholder needs in a timely manner

3. Provide a mostly automated tool that utilizes AI/ML within the decision-making process for the other goals above

The remainder of this dissertation is as follows:

- **Chapter 2:** We dive into the basic foundations behind IC-GSBD. We begin with the basics of Set-Based Design (SBD), its purpose and workflow. Then, we discuss graph theory by discussing what graphs are and how they are represented mathematically, followed by how systems can be represented as graphs. Then, we discuss Geometric Deep Learning and Graph Neural Networks for the AI/ML component.

- **Chapter 3:** Here, we describe the methodological framework of IC-GSBD. We begin with an in-depth discussion of graph classification, covering both binary and multi-label classification methods. Following that, we dive into feature engineering with a discussion of the process and how it benefited IC-GSBD with an example. Next, we discuss the model and its corresponding layers used for the classification process, followed by the hyperparameters and the metrics used to determine model performance. We then discuss iterative classification from dataset preparation through the algorithm. Lastly, we discuss GDL's and iterative classification's role in GSBD.

- **Chapter 4:** In this chapter, we go into the first case study, the classification and down-selection of analog electric circuits. Here, we see the basics of IC-GSBD and how it can take a large dataset of 43,000+ graphs and narrow it down to the top graphs in the set.

- **Chapter 5:** Here, we look at the second case study, the classification and down-selection of aircraft thermal management systems (TMSs). Additionally, we not only demonstrate narrowing down a graph-based dataset to the top graphs in the set but also demonstrate a multi-label classification approach.

- **Chapter 6:** Lastly, we summarize the work with key points and future work for IC-GSBD and related engineering design problems.

Chapter 2

Background[1]

> "The strength of a building lies in its foundation. So, too, does the strength of any engineering project lie in its initial design and planning."
>
> — *Henry Petroski* [53]

The Iterative Classification for Graph-Set-Based-Design (IC-GSBD) methodological framework that will be defined in the next chapter has many aspects that require discussion and understanding in order to fully grasp its effectiveness as a tool. This chapter begins with the basics of Set-Based Design (SBD), which lays some of the foundations for GSBD. Following this, we delve into graph theory and discuss what graphs are and how they are mathematically represented, which is essential for understanding the graph-based nature of IC-GSBD. We then explore how systems can be represented as graphs, providing a bridge between theoretical graph concepts and their practical applications in system design. Since IC-GSBD utilizes extensive graph datasets, we introduce the method of graph enumeration, which was used to generate the vast sets of graphs in the case studies presented in this dissertation. Lastly, we explore Geometric Deep Learning (GDL) and discuss the various tasks that GDL can accomplish utilizing Graph Neural Networks (GNNs). We will also review some example model architectures to illustrate the practical implementation of GDL within the IC-GSBD methodological framework.

2.1 Set-Based Design (SBD)

Set-Based Design (SBD) is a transformative design methodology that underscores the importance of exploring a wide array of potential solutions at the outset of the design process. Unlike traditional Point-Based Design (PBD), which focuses on iteratively refining a singular design solution within a predefined set of constraints, SBD facilitates a comprehensive exploration of the

[1] Parts of this chapter are based on Refs. [51, 52]

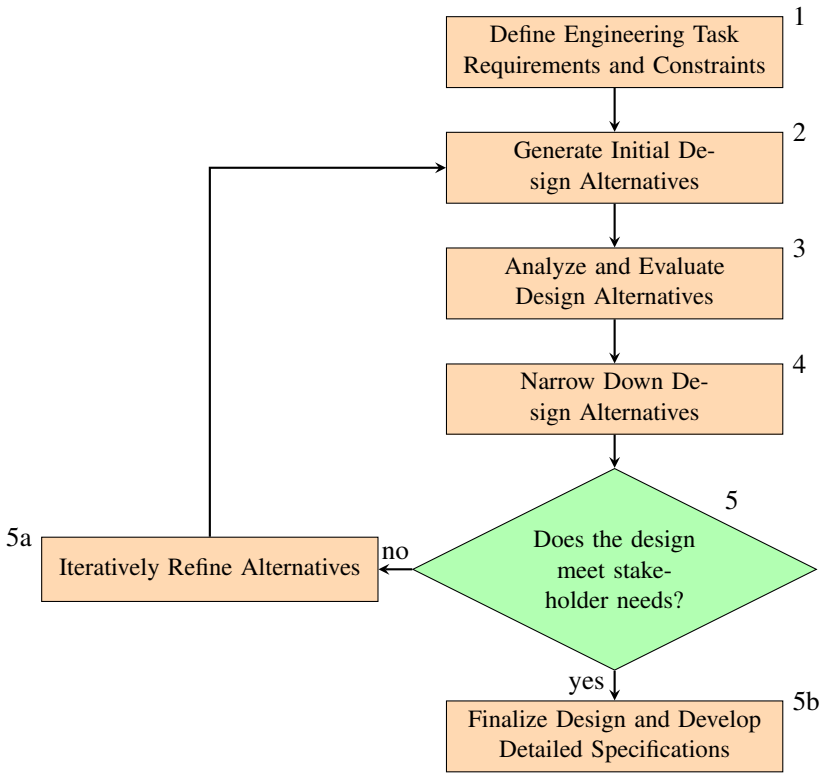


Figure 2.1: The workflow for Set-Based Design

design space. This approach enables designers to concurrently consider multiple alternatives and their interrelations, fostering innovation and leading to more optimized solutions [54, 55].

Historically, SBD began in the automotive and aerospace industries. There was a need for an efficient and innovative design process, and over time, SBD evolved and was adopted in various engineering fields, which also highlights its versatility and effectiveness. Utilizing an SBD framework presents several advantages, including concurrent consideration of alternatives, which encourages the exploration of diverse design solutions simultaneously. SBD gives designers the ability to identify constraints and requirements by evaluating a wide range of solutions to help identify them early in the design process. By allowing for an in-depth assessment of trade-offs between different design alternatives, SBD enhances the decision-making process. Lastly, SBD reduces design iterations and mitigates the risk of prematurely converging on a single design solution, potentially reducing the number of iterations needed to reach an optimized design.

As highlighted in Ref. [14, 56, 57], the SBD process encompasses a structured approach to engineering design as shown in Figure 2.1. Initially, it involves defining the requirements and constraints that shape the exploration of potential design solutions (Step 1), ensuring all considered alternatives align with the project's overarching goals. Following this, a wide array of design alternatives is developed (Step 2), deliberately avoiding early commitment to any single solution. Subsequently, parallel analysis and evaluation of these alternatives are conducted against the set requirements and constraints (Step 3), a step underscored for its importance in iterative refinement and decision-making. This analysis helps in progressively narrowing down the options based on their feasibility and alignment with project goals. (Step 4). The narrowing process is inherently iterative, requiring repeated analysis and evaluation to continually refine the set of alternatives until a final design is selected (Step 5). This chosen design then undergoes the development of detailed specifications (Step 5b), marking the transition from a conceptual stage to detailed engineering and production planning. Alternatives are iteratively refined if the design does not meet stakeholder needs (Step 6).

Real-world applications of SBD include its use in the development of the Toyota Prius, where multiple design alternatives were explored concurrently to optimize fuel efficiency and performance [14]. In the aerospace industry, SBD has been instrumental in designing complex systems with stringent performance and safety requirements [57].

We have shown the advantages of SBD, but it is not without its challenges. Implementing SBD can require significant upfront investment in time and computational resources to generate and evaluate large sets of design alternatives. Additionally, it demands high collaboration and communication between multidisciplinary teams to effectively analyze and narrow down options [54]. Therefore, as effective as SBD is, research methods that can increase computational savings are still needed. In Section 3.7, we will discuss how GSBD utilizes the fundamentals of SBD. But first, in order to understand these graph-based techniques, we will dive into the basics of graph theory in the next section.

2.2 Graph Theory

Graph theory is the core foundation for this approach, as it encapsulates the dimensions and features of the system in question. A *graph* has various names, such as a node-link chart, network map, or link diagram, but for the purposes of this work, we will refer to it simply as a graph. In this section, we will discuss the fundamentals of graph theory, beginning with the basic definitions of graphs and how they are represented mathematically. We will also discuss the information that can be extrapolated from graphs, followed by a discussion of how we can represent various systems as graphs. We will work with two types of graphs within these case studies: *undirected* (Case Study 1) and *directed* (Case Study 2). Let us begin with some definitions:

Definition 1 (Undirected Graph). A *graph* G is a pair of sets (V,E) where $E \subseteq [V]^2$ (i.e., E is a two-element subset of V). E represents the edges of the graph, while V is the set of its vertices or nodes [58]. This is represented in Figure 2.2.

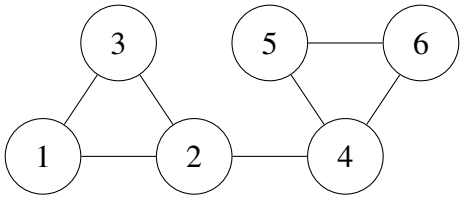


Figure 2.2: A simple undirected graph

Definition 2 (Directed Graph). A directed graph, or digraph, is a pair (V, E) of disjoint sets together with two maps init:$E \to V$ and ter:$E \to V$ assigning to every edge e an initial node init(e) and a terminal node ter(e) [58], shown in Figure 2.3.

This orientation arises when one node acts as a "source" node and is directed towards another. If we examine Figure 2.3, it becomes apparent that in a directed graph, connections are represented by directed arrows instead of lines, indicating the direction of flow from the source node.

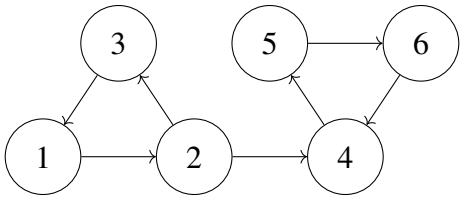


Figure 2.3: A simple undirected graph

There are many ways to represent a graph mathematically, the first being an *incidence matrix* $\mathbf{B} = (b_{ij})_{n\times m}$ fo a graph $G = (V, E)$ with $V = \{v_1, ..., v_n\}$ and $E = \{e_1, ..., e_m\}$ is defined over $\mathbb{F}_2$ by,

$$b_{i,j} \coloneqq \begin{cases} 1, & \text{if } v_i \in e_j \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

where $\mathbb{F}_2$ is defined as a finite field with two elements, typically denoted as $\{0, 1\}$ [58].

Another way of representing a graph mathematically is what is known as an *edge list*. An edge list can be defined as,

$$E\{(v_i, v_j)|v_i, v_j \in V \tag{2.2}$$

for an undirected graph G, where each edge $e \in E$ is represented as $e = (v_i, v_j)$, and (v_i, v_j) indicates that there is an edge between vertices v_i and v_j [59]. Similarly, there is also what is known as an *adjacency list*, which is a list of the neighbors of vertex v denoted as $N_G(v)$, and a list of these lists is the adjacency list $(N(v) : v \in V)$ [60].

One other way is what is known as the *adjacency matrix* $\mathbf{A} = (a_{i,j})_{n\times n}$ and is defined as:

$$a_{i,j} \coloneqq \begin{cases} 1, & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

where n represents the number of nodes in the graph and the pair (i, j) represents the row and column node indices, respectively, in the matrix. Equation (2.3) is used for both directed and undirected graphs, the difference being that for directed graphs, the corresponding adjacency matrix is not symmetrical, $a_{i,j} \neq a_{j,i}$. The adjacency matrix will be the primary notation that we will focus on for the remainder of this dissertation.

Beginning with the undirected graph in Figure 2.2, using the adjacency matrix notation, it would be denoted as,

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \\ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad = \mathbf{X} \tag{2.4}$$

where, as shown in Equation (2.3), a 1 represents a connection. For example, $a_{1,2} = 1$, where the 1 indicates that node 1 (row 1), has a connection with node 2 (column 2), as indicated in Figure 2.2. Since an undirected graphs' adjacency matrix is symmetrical, $a_{i,j} = a_{j,i}$, you can determine node connections beginning with column first as well.

Similarly, using the directed graph in Figure 2.3, its corresponding adjacency matrix is denoted as,

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \\ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad = \mathbf{X} \tag{2.5}$$

As you can see, both graphs have the exact same structure, but because this one is directed, its corresponding adjacency matrix only has a 1 where that node is a source node. Therefore, $a_{1,2} = 1$ indicates that node 1 is a source node for node 2, but since the matrix is not symmetrical, $a_{2,1} \neq 1$

as it does in Equation (2.4). Lastly, in the case of the last row and column, here they represent the corresponding node number, but it other cases, they are reserved for the node features $\mathbf{X}$.

Another important concept is graph isomorphism. Consider two graphs, $G = (V, E)$ and $G' = (V', E')$ shown in Figure 2.4. These graphs are termed isomorphic, expressed as $G \cong G'$, if a bijection φ exists from V to V' such that for every pair of vertices (v_i, v_j) in E, the corresponding pair $(\varphi(v_i'), \varphi(v_j'))$ is in E'. In other words, both graphs G and G' have the same structure but are labeled differently. To show that these two graphs are isomorphic, we can map the vertices of G to G', $v_1 \leftrightarrow v_1'$, $v_2 \leftrightarrow v_3'$, and $v_3 \leftrightarrow v_2'$. This is important to take note of because when it comes to designing systems and representing them as graphs, a designer will want a dataset of *unique* (non-isomorphic) graphs.

(a) G **(b)** G'

Figure 2.4: Two isomorphic graphs

Two other concepts to discuss in graph theory are the features matrix $\mathbf{X}$ and weighted edge matrix $\mathbf{W}$. The features matrix, represented as,

$$\mathbf{X} \in \mathbb{R}^{n \times c} \tag{2.6}$$

can hold a plethora of information, whereas before, n represents the number of nodes, but c is the number of features each node has.

The weighted edge matrix is a matrix where each element $w_{i,j}$ represents the weight of the edge between vertices v_i and v_j and is mathematically represented as,

$$w_{i,j} \coloneqq \begin{cases} \text{weight}(v_i, v_j), & \text{if } (v_i, V - j) \in E \\ 0, & \text{otherwise} \end{cases} \tag{2.7}$$

These matrices are crucial for capturing the complexities and attributes of graph-structured data, enabling sophisticated analyses and manipulations. They will be elaborated upon further in the next section, where we discuss representing systems as graphs.

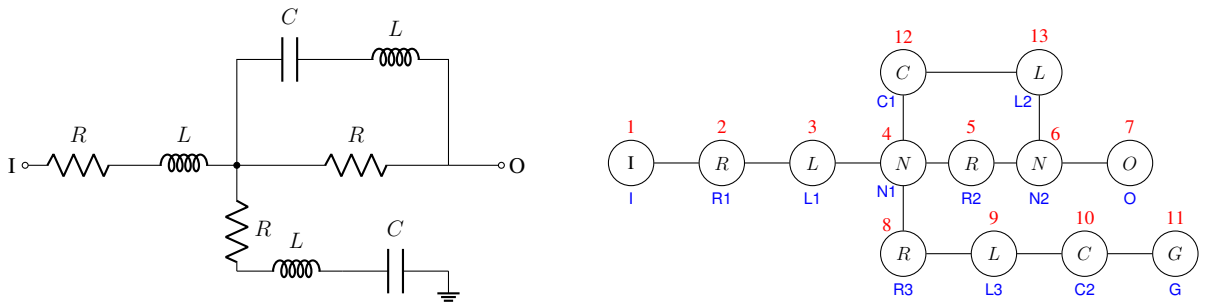
Continuing the discussion on isomorphism, using the same two graphs, if they include an additional feature such as a vertex label, making the graphs $G = (V, E, X)$ and $G' = (V', E', X')$, they are considered isomorphic if this vertex label characteristic is maintained under a valid bijection φ.

With these foundational concepts of graph theory established, we can now transition to representing systems as graphs. This approach is crucial for modeling and analyzing complex systems within the IC-GSBD framework.

2.2.1 Representing Systems as Graphs

The concepts from the previous section can be utilized to model engineering systems. We will give two examples from the case studies used in this work: an analog RLC circuit and an aircraft thermal management system.

RLC stands for Resistor, Inductor, and Capacitor; a simple example of one of these circuits is shown in Figure 2.5. Looking at Figure 2.5a, we see a basic RLC circuit schematic. Then, in Figure 2.5b, we have the same circuit represented as a graph. In the graph, each vertex is labeled with the corresponding component, where I and O represent the input and output nodes, and N represents a voltage node with constant voltage. It's also worth noting that other possible graph representations of the same circuit exist. To represent this same circuit mathematically, we begin



(a) RLC Circuit **(b)** Graph

Figure 2.5: A simple RLC circuit represented as a graph

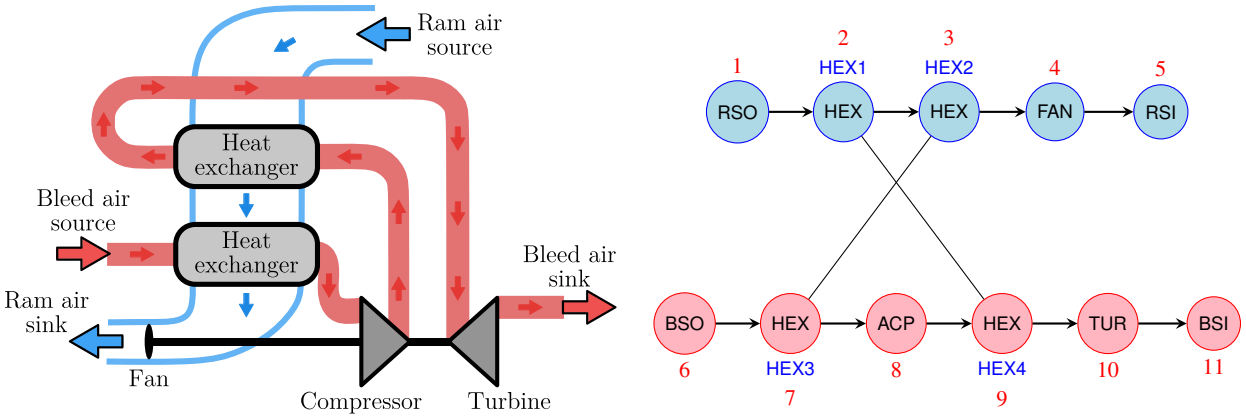
with Equation (2.3),

$$\mathbf{A} = \begin{bmatrix} 0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&1&0&0&0&0&0&1&0 \end{bmatrix} \begin{matrix} \text{I} \\ \text{R1} \\ \text{L1} \\ \text{N1} \\ \text{R2} \\ \text{N2} \\ \text{O} \\ \text{R3} \\ \text{L3} \\ \text{C2} \\ \text{G} \\ \text{C1} \\ \text{L2} \end{matrix} \quad (2.8)$$

$$\text{I}\ \ \text{R1}\ \ \text{L1}\ \ \text{N1}\ \ \text{R2}\ \ \text{N2}\ \ \text{O}\ \ \text{R3}\ \ \text{L3}\ \ \text{C2}\ \ \text{G}\ \ \text{C1}\ \ \text{L2} = \mathbf{X}$$

as you can see, it gives us a 13×13 symmetrical matrix. You may notice that there is an extra row and column, which includes the *features* for the graph using Equation (2.6). This particular features matrix is 13×1 since there is only one feature, the node label. But, we can include more information if so desired, such as the values of the components.

Now, we will take a look at the TMS and how it is represented as a graph. This one is a bit more difficult to distinguish, but the same concept applies. The components you see are the Ram Air Source (RSO), Heat Exchanger (HEX), Fan (FAN), Ram Air Sink (RSI), Bleed Air Source (BSO), Air Compressor (ACP), Turbine (TUR), and Bleed Air Sink (BSI). Once again, using Equation (2.3), remembering that this is now for a directed graph where a 1 indicates that the corresponding com-



(a) Illustration **(b)** Graph representation

Figure 2.6: A common air cycle machine (ACM) architecture represented in two forms

ponent (row) is a source for the column component, and we end up with Equation (2.9).

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} \text{RSO} \\ \text{HEX1} \\ \text{HEX2} \\ \text{FAN} \\ \text{RSI} \\ \text{BSO} \\ \text{HEX3} \\ \text{ACP} \\ \text{HEX4} \\ \text{TUR} \\ \text{BSI} \end{matrix} \quad (2.9)$$

$$\text{RSO}\ \ \text{HEX1}\ \ \text{HEX2}\ \ \text{FAN}\ \ \text{RSI}\ \ \text{BSO}\ \ \text{HEX3}\ \ \text{ACP}\ \ \text{HEX4}\ \ \text{TUR}\ \ \text{BSI} = \mathbf{X}$$

One thing to note with this adjacency matrix: if we observe $a_{2,2} = 1$ for HEX1, it has an additional 1. This indicates that it has what is known as a *self-loop*. If we go back and look at Figure 2.6a, we can see that the system has two heat exchangers, but they are each used for both the bleed air system and the ram air system. Looking at Figure 2.6b, you will also notice four heat exchangers. That is because for each of the two heat exchangers, there is a node that represents which part of the overall TMS it belongs to, and since HEX1 and HEX4 are connected as well as HEX2 and HEX3 to show their relationships, we give one heat exchanger from each pair the self-loop.

These examples illustrate how some systems can be modeled as graphs, capturing the relationships and interactions between various components. The RLC circuit and the aircraft TMS demonstrate the versatility and applicability of graph representations in different domains. By representing systems as graphs, we can leverage powerful mathematical tools and algorithms for analysis, optimization, and design.

Having established the importance and methodology of representing systems as graphs, we now turn our attention to the next crucial step in our framework: graph enumeration. This method is fundamental for generating the extensive sets of graphs required for the IC-GSBD framework, facilitating the structured and systematic exploration of vast design spaces.

2.2.2 Graph Enumeration

Designers and system architects still often rely on engineering intuition and trial-and-error methods when exploring graph-centric problems. However, these approaches can lead to a lack of novel and satisfactory solutions within time restrictions.

A potential systematic strategy is *graph enumeration*, which are techniques for generating (or sometimes simply counting) non-isomorphic graphs with particular properties [61]. The properties can be quite diverse and are sometimes termed network structure constraints [34]. Some examples include bounding the potential degrees of the vertices in the graph or a maximum cost associated with the graph [28]. It is important to note that the number of valid graphs is usually combinatorial in nature; thus, the computational costs for just generating an enumeration can become quite expensive. However, two different graph enumeration algorithms might produce the desired enumeration, but one might do it more efficiently [28].

In any case, there are several important engineering applications where the enumeration of graphs, often with thousands to millions of entries, is practical. Even if graph enumeration is not possible, generating a partial listing (either algorithmically or manually) may still be desirable to explore potential solutions. In this work, it is assumed that a generated set of graphs denoted $\mathcal{G}$, contains the graphs of interest.

While having a list of novel graphs can be useful on its own, in many engineering domains, the graph is only an intermediary representation used to compute one or more value metrics of interest. For example, the graph might correspond to a physics-based system model that is used to determine the comfort and handling of an automotive suspension [62]. For this work, we will consider a single performance metric, denoted $J(G_i)$, that is a function of the graph, and the natural goal is seeking graphs that minimize this "performance" metric:

$$\underset{G_i}{\text{minimize:}} \quad J(G_i) \tag{2.10}$$

One of the challenges in engineering design with graphs is sometimes the computational cost of $J(G_i)$ is quite expensive, perhaps many orders of magnitude larger than generating the graph G_i itself. The source of this cost can be quite diverse, including high-fidelity simulations, optimization, human-centric evaluation, and physical experiments. The focus of this work is when this cost is prohibitive.

Based on the classification put forth by [63], we define the following three types of graph-centric design problems:

- *Type 0* — All desired graphs can be generated and so can their performance metric $J(G_i)$ within time T
- *Type 1* — All desired graphs can be generated, but only some of the performance metrics $J(G_i)$ can be evaluated within time T; the performance assessment is too expensive
- *Type 2* — All desired graphs cannot be generated within time T

where T is the amount of time allocated to complete the graph design study. This work focuses on methods for Type 1 problems (using data from a large Type 0 study).

Graph enumeration provides a systematic approach to generating and exploring extensive sets of graphs, essential for identifying novel and optimized design solutions. However, the computational cost of evaluating performance metrics for these graphs can be prohibitive for complex systems. This is where advanced machine learning techniques, specifically Geometric Deep Learning (GDL), come into play. GDL leverages the structure of graph-encoded data to perform efficient

and scalable analyses, making it an invaluable tool in addressing the challenges of graph-centric design problems. In the next section, we will explore Geometric Deep Learning and discuss how it can be utilized within the IC-GSBD framework to enhance the design and optimization process.

2.3 Geometric Deep Learning (GDL)

Now that we have discussed the basics of graph theory in the previous section, we can see how they are used and manipulated within a machine-learning model. Graphs fundamentally differ from the more common Euclidean data in most deep-learning applications (e.g., images, text, and speech); Euclidean data has an underlying grid-like structure [64–66]. For example, an image can be translated to an (x, y, z) Cartesian coordinate system, where each pixel is located at an (x, y) coordinate, and z represents the color. But, this approach only works for some problems because certain operations require many dimensions. This "flat" representation has limitations, such as being unable to represent hierarchies and other direct relationships.

These issues motivated the study of hyperbolic space for graph representation, or non-Euclidean learning [67, 68], which could potentially perform those same operations more flexibly. For example, many real-world data sets are naturally structured as graphs, where the *relationships* between data points or other entities are integral. Euclidean-based methods struggle with such data, as they are designed to operate on flat, unstructured data. On the other hand, GDL methods can directly exploit the data structure, leading to better outcomes [36].

Since its inception, Geometric Deep Learning (GDL) can be applied to many forms of data represented as geometric priors [69]. Geometric priors encode information about the geometry of the data, such as smoothness, sparsity, or the relationship between the data and other variables, allowing us to work with data of higher dimensionality. Symmetry is essential in GDL and is often described as invariance and equivariance. Invariance is a property of particular mathematical objects that remain unchanged under certain transformations. At the same time, equivariance is a property of certain relations whereby they remain in the same relative position to one another

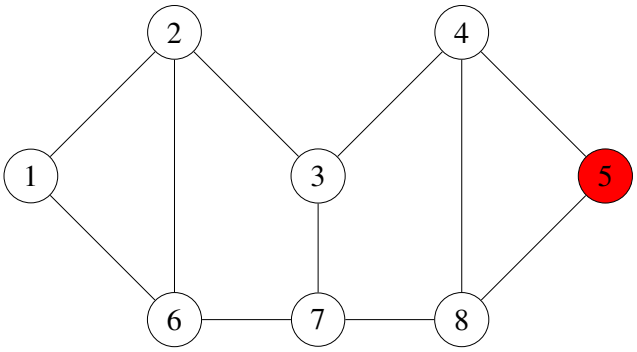


Figure 2.7: Anomaly detection

under specific changes [70, 71]. For example, consider the graph isomorphism property from Section 2.2.2. Another advantage of GDL is its flexibility toward a broader range of data types and problems. This flexibility makes it a powerful tool for solving problems that Euclidean methods cannot.

2.3.1 GDL Tasks

There are various tasks that leverage the power of GDL's ability to process and analyze non-Euclidean data:

- *Anomaly Detection* (Figure 2.7) is a binary classification method that identifies unusual patterns within a dataset that significantly deviate from the majority of the data. The objective is to detect these anomalies, which could indicate errors, fraud, or other atypical events, and subsequently flag them [72, 73].

- *Node Classification* (Figure 2.8) involves assigning labels to nodes in a graph based on their properties and the relationships between them [74, 75].

- *Link Prediction* (Figure 2.9) aims to predict missing connections between nodes in a network. Given a partial network, the goal is to infer which links are most likely to be added or are missing based on the observed connections and the overall network structure [76, 77].

- *Graph Classification* (Figure 2.10) involves classifying graph-structured data into various categories. The input is a graph, and the goal is to train a classifier that can accurately

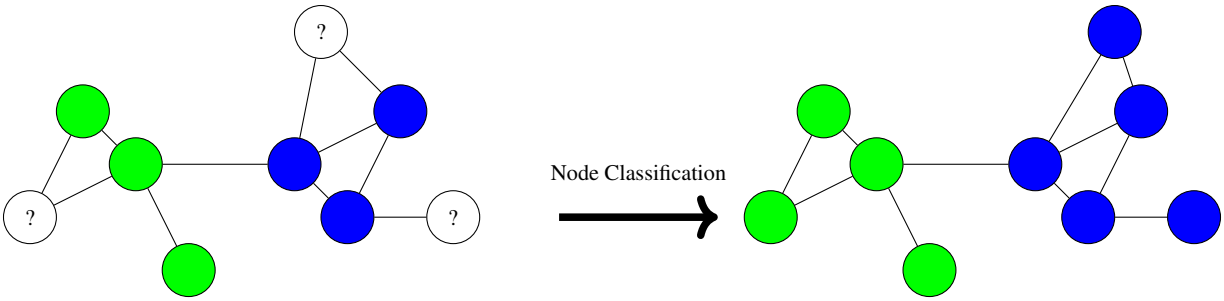


Figure 2.8: Node classification

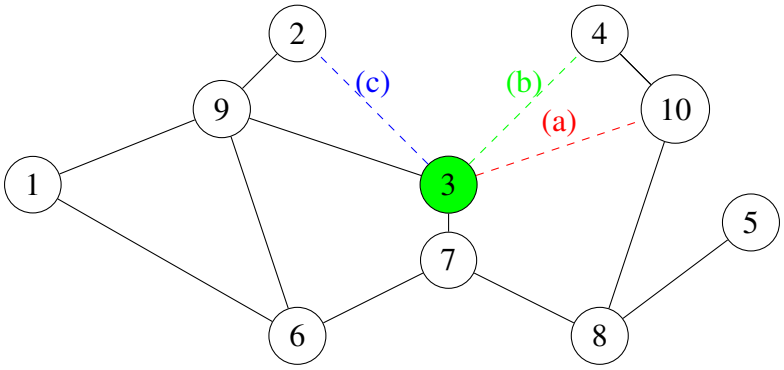


Figure 2.9: Link Prediction

predict the class of the graph [51, 52, 64, 78]. An example of graph classification shown in Figure 2.10 shows two classes (categories) for the graph, good or bad, based on predefined criteria. Graph classification is the primary focus of this work and is discussed further in Section 3.1.2.

2.3.2 Graph Neural Network (GNN)

GDL heavily relies on the Graph Neural Network (GNN) as one of its primary tools. This type of network is considered to be among the most versatile classes of deep learning architectures currently available. What sets GNN apart is its ability to encompass other deep learning architectures as special cases, especially when additional geometric structure is introduced. This flexibility and adaptability make GNN a powerful and fundamental tool in the field of geometric deep learning. First, let us define what a neural network is. A Neural Network is a machine learning program or model that makes decisions in a manner similar to the human brain by using processes that mimic

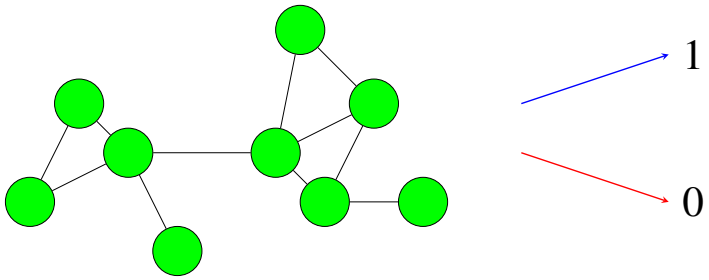


Figure 2.10: Graph classification

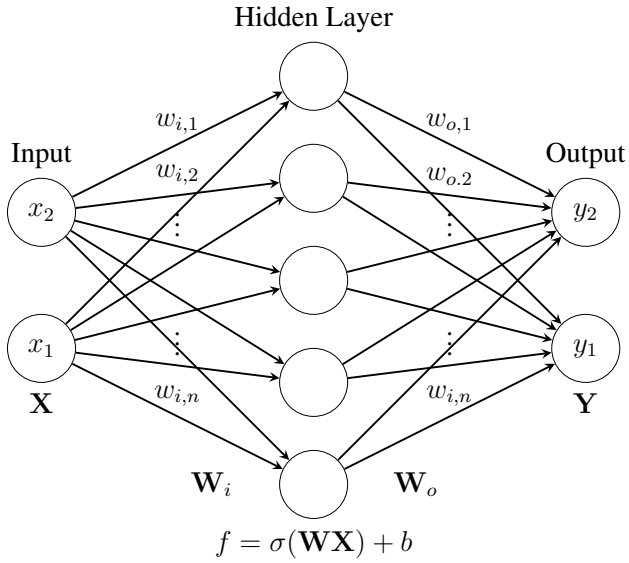


Figure 2.11: A basic neural network

the way biological neurons work together to identify phenomena, weigh options, and arrive at a conclusion [79].

The basic element of a neural network is a neuron, and a neural network is derived by connecting the neurons into a structured layer, sometimes referred to as a hidden layer. A basic neural network is depicted in Figure 2.11. The model receives signals from the input neurons via weighted connections ($w_{i,n}$). The weighted sum of the received signals is compared against the threshold, or known bias, and the output signal is produced by the activation function such as Rectified Linear Unit (ReLu) or sigmoid (σ) [80].

A deep learning network typically comprises multiple hidden layers, usually four or more. In Figure 2.12, the network is shown to have four hidden layers. Each hidden layer ($h_1, h_2, \cdots, h_n$)

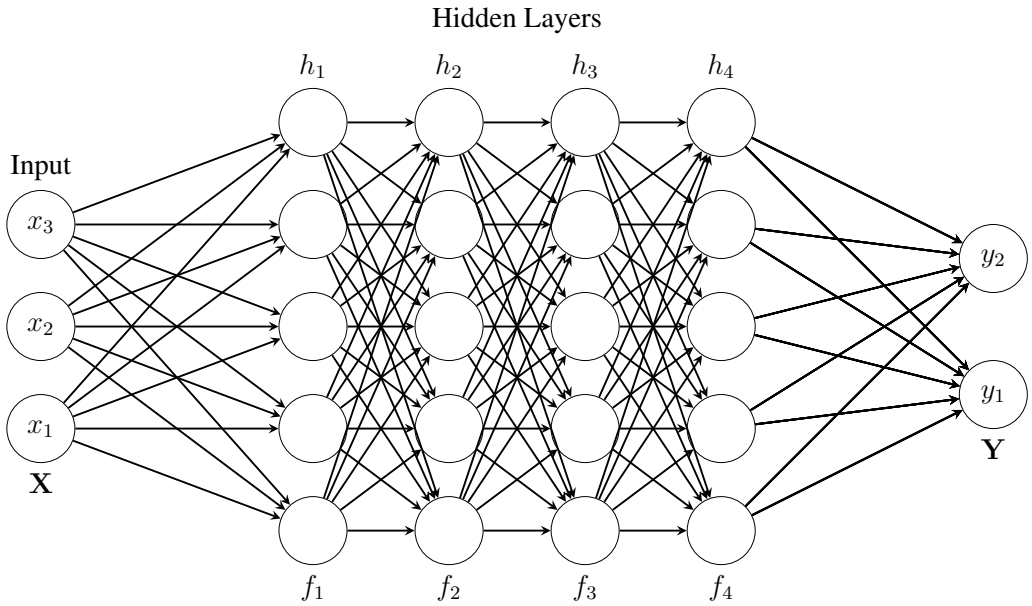


Figure 2.12: A deep learning neural network

includes an activation function, enabling the network to simulate the complex decision-making capabilities of the human brain [81].

Similar to traditional deep learning models, Geometric Deep Learning utilizes GNNs, which are used to learn representations of graph-structured data, such as social networks, molecules, and computer programs. Some deep learning models are what is known as a Convolutional Neural Network (CNN). A CNN is a feedforward neural network that is able to extract features from the input data with convolutional structures [82]. A GNN is very similar to a CNN in that it can extract localized features and compose them to construct representations [83]. The critical difference between the two is that GNNs can learn on non-Euclidean data and handle data that is not evenly structured, like images or text.

2.3.3 Basic Principles

Training a GNN for graph classification, which involves determining whether a graph possesses a specific characteristic, typically follows a four-step process. The first step is initialization, where each node in the graph is represented by a feature vector, often based on the properties of the node. The second step is message passing, during which nodes in the graph exchange information

with neighboring nodes through a learnable function that combines the features of the neighboring nodes. In the third step, the feature vectors of the graph nodes are updated by aggregating the messages received from their neighbors. Finally, after a predetermined number of iterations, the final representations of the graph nodes are utilized for downstream tasks such as graph classification. In this step, the representations are passed through a conventional layer, such as a linear layer [84]. The choice of specific layers for a GNN is a critical decision; the next chapter will explore this in detail.

2.3.4 Example GNN Models and Methods

GNNs have been developed in various architectural forms to address the diverse challenges presented by graph-structured data. These models leverage different methods for message passing, aggregation, and feature transformation to optimize performance for specific tasks. Here, we will explore several types of GNN models and methods that have demonstrated significant efficacy in a range of applications. Each one offers unique mechanisms for processing and learning from graphs, showcasing the versatility and adaptability of GNNs in handling complex relational data.

Deep Graph Convolutional Neural Network (DGCNN)

By taking graphs as a direct input, this architecture has the ability to keep more vertex information and learn from the global graph topology. It proposes a spatial graph convolution layer that extracts multi-scale vertex features and passes them through a SortPooling layer, which takes the graph's unordered vertex features, arranges them in a consistent order, and outputs a graph representation [84].

GraphSAGE

GraphSAGE, which stands for Graph Sample and Aggregate, was introduced for inductive node embedding. It leverages node features to learn an embedding function that generalizes to unseen nodes. By doing so, the architecture can learn the topological structure and distribution of

node features in each node's neighborhood. But, not all graphs have features, so this architecture can also use the structural features that are within the graph [85].

EigenPooling

EigenPooling is a method that incorporates a new operator introduced to an existing model. This operator tackles the challenges encountered by general pooling operators, such as incorporating local structure information when the subgraphs contain different numbers of nodes and very different structures. EigenPooling is based on the eigenvectors of the subgraphs, which capture the local structures of the subgraphs because the eigenvectors have the same size for each subgraph [86].

DiffPool

Another method that is applied to an existing model, DiffPool is a differentiable graph pooling module that can generate hierarchical representations of graphs and can be combined with various GNN models [87].

2.4 Conclusion

This chapter discussed the foundational concepts that the IC-GSBD methodological framework is built upon. We began by exploring Set-Based Design (SBD), highlighting its structured approach to considering multiple design alternatives simultaneously and progressively narrowing down to the optimal solution. This method's inherent flexibility makes it an essential precursor to our proposed framework.

We then transitioned into graph theory, highlighting how systems can be effectively represented as graphs. Graph representations provide a powerful tool for modeling complex systems by capturing the intricate relationships and dependencies between components. This representation is crucial for IC-GSBD, as it efficiently uses graph-based methods to explore and evaluate a vast design space.

Finally, we introduced Geometric Deep Learning (GDL), a broader class of machine learning techniques that generalize neural networks to non-Euclidean domains. GDL's capacity to incorporate information about the structure space and symmetry properties of input variables enhances the quality of data representations, making it a crucial component of the IC-GSBD framework.

By integrating these concepts, the IC-GSBD framework offers a novel and efficient system design and optimization approach. It combines the exploratory power of SBD with the predictive capabilities of GDL, facilitating a more informed and dynamic decision-making process. This synthesis enables the handling of complex design problems with greater precision and efficiency, allowing for innovative solutions in engineering design. As we move forward, the subsequent chapters will dive deeper into the methodological framework of IC-GSBD, explore its application through case studies, and demonstrate its effectiveness in various engineering contexts.

Chapter 3

Iterative Classification and Graph-Set-Based Design Methods[2]

> "The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill. To raise new questions and possibilities and regard old problems from a new angle requires creative imagination and marks real advances in science."
>
> — *Albert Einstein* [88]

In the previous chapters, we covered the broader subjects that build the foundation of Iterative Classification (IC) and Graph-Set-Based Design (GSBD). Now, it is time to see how these concepts come together. This chapter discusses the detailed aspects of GSBD, providing a comprehensive understanding of its methodology and application.

We begin with an in-depth discussion of graph classification and its employment in the two case studies, starting with binary classification and then moving on to multi-label classification. Next, we explore feature engineering, including the use of Principal Component Analysis (PCA) and its benefits to data analysis. Following this, we review the specific model used in IC-GSBD, examining the hidden layers chosen, the rationale behind their selection, and the model's hyperparameters. We also discuss the various metrics used to evaluate model performance. We then introduce the premise of iterative classification and, finally, discuss the role of Geometric Deep Learning (GDL) in set-based design. This chapter aims to provide a detailed and technical perspective on GSBD, highlighting its innovative approach to complex system design.

[2]Parts of this chapter are based on Refs. [51, 52]

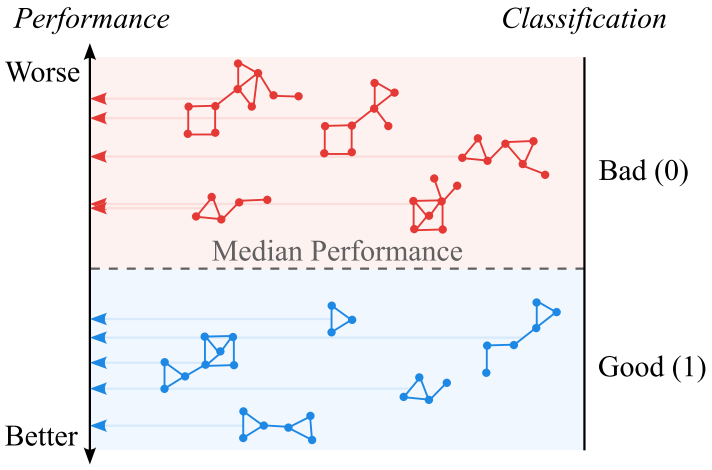


Figure 3.1: Illustration of graph classification based on performance values.

3.1 Graph Classification

In Section 2.3.4, we briefly introduced the concept of graph classification. Here, we will dive deeper into the rationale for selecting this method and its application in our study.

Graph classification refers to classifying graphs based on selected labels. Here, we consider *supervised* graph classification, where we have a collection of graphs in set $\mathcal{G}$, each with its label $J(G_i)$, in this case, a specific value or metric used to determine the performance or ranking of the graphs within the set. This data is used to train a model using additional graph properties, such as the structure, embeddings, and node data. These features help the model discriminate between the graphs, improving the prediction of the labels.

Graph classification is simplistically illustrated in Figure 2.10 in Section 2.3.4. Figure 3.1 shows a little more detail, whereas how it is done in this study. Here, the median performance value on the left axis might be used as the determination point to divide our graphs into binary classes representing "good" graphs in the lower half of the observed performance values and "bad" graphs in the upper half.

The rationale for this approach over regression was to better reflect the designer's intention in early-stage conceptual design (i.e., flexible down-selecting). Often, the goal is not to narrow the potential graphs down to *one* particular graph, but rather a *subset* of "good" or promising graphs that would be analyzed further. This approach allows for flexibility in down-selecting a group of

"good" graphs for further analysis at a higher fidelity, taking into account the assumptions made during the initial graph design phase.

Furthermore, the performance values $J(G_i)$ in the case studies have significant variations and many coarse and repeated values. Therefore, the intent is to show the graphs' "belongingness" to a specific group rather than assigning specific values to them. However, a regression-based approach could be used as the basis for the down-selection task if such a model can perform well at discerning the set of desirable graphs (and would come with the additional benefit of a performance prediction). However, previous work using regression on these datasets was not particularly performant [63].

Various techniques have been developed for graph classification. These advanced methods have been applied in diverse domains, including:

- **Molecular Biology**: Graphs representing molecules are classified to predict biological activity based on molecular structure [89].
- **Text Categorization**: Graphs representing documents are categorized to predict document classes [90].
- **Computer Networks**: Graphs representing network traffic are analyzed to detect patterns indicative of malicious activity [91].
- **Cancer Research**: Graphs representing protein-protein interaction networks are classified to identify potential drug targets [92].

By leveraging these techniques, graph classification enables the effective analysis and categorization of complex graph-structured data, offering valuable insights across various scientific and practical applications. In this study, we use two types of classification: 1) binary classification and 2) multi-label classification.

3.1.1 Binary Classification for Best Graph Performance

This technique is used in both case studies. In the first case study, some median performance value $\bar{J}$ is a benchmark for categorizing the graphs into two binary classes, where a smaller value $J(G_i)$ is considered better or more desirable. Graphs falling in the lower half of the observed performance values (i.e., $J(G_i) \leq \bar{J}$) are, therefore, labeled as "good" and assigned the label 1, while those in the upper half (i.e., $J(G_i) > \bar{J}$) are considered "bad" and assigned the label 0.

In the second case study, we had a similar scenario. However, this time $J(G_i)$ represented a utility value determined by different aspects of the thermal management system, which is further explained in Chapter 5. The same principle applies here: we categorize the graphs as desirable or undesirable based on the median utility value.

This method is highly beneficial for swiftly narrowing down to the most desirable graphs. When used in conjunction with IC, $\bar{J}$ will keep changing based on the specific task at hand, whether it involves seeking a smaller or larger value of J.

3.1.2 Multi-Label Classification for Multiple Graph-Level Features

This is another beneficial technique used in situations with multiple graph-level features and is one of the techniques used in the second case study in Chapter 5. When we have multiple graph labels, we can represent them for graph G_i as:

$$\mathbf{L}(G_i) = [l_1, l_2, \ldots, l_{c-1}, l_c] \tag{3.1}$$

where in this label array, each l_c represents a different aspect or *sub-label*. The rationale for adopting a multi-label strategy is to be able to answer several different questions for each graph using a single model instead of requiring multiple models for other questions.

For the area of interest in the second case study in Chapter 5, we are interested in answering two particular questions about each graph,

1. Can the graph be compiled (i.e., can a physics-based model be automatically constructed from a given graph)?

2. Is the graph capable of being simulated (i.e., given a complied model, are there valid simulation outcomes)?

Based on the answers to these questions, we derive four possible categories for the graphs: graphs that will not compile, graphs that will compile, graphs that will not simulate, and graphs that will simulate.

Now, each graph will have the label array $\mathbf{L}(G_i) = [l_1, l_2, l_3, l_4]$. The subset $[l_1, l_2]$ corresponds to the graph's ability to compile (or not), and $[l_3, l_4]$ relates to its ability to simulate (or not). The three classification options are then:

$$\mathbf{L}(G_i) = \begin{cases} [0,1,0,1] & \text{if } G_i \text{ will compile and simulate} \\ [0,1,1,0] & \text{if } G_i \text{ will compile but not simulate} \\ [1,0,1,0] & \text{if } G_i \text{ will not compile} \end{cases} \tag{3.2}$$

Graph classification is a powerful tool that engineers can utilize, whether to narrow down a large set of graphs to the subset of more promising graphs or to answer a set of questions based on specific features of the graph. The next section discusses ways to extrapolate features from the graph when other useful features are absent. This is known as feature engineering and is a useful tool when combined with Principle Component Analysis (PCA).

3.2 Feature Engineering

In this context, we explore augmenting the feature set $\mathbf{X}$ with additional graph-based attributes as described in Section 2.2.2 extending beyond the existing vertex labels such as R, L, and C from the first case study or HEX, BSO, and BSI from the second. Incorporating these supplementary features can enhance the model's capacity to account for the variance within the training data, a practice commonly referred to as feature engineering and feature selection. The underlying hypothesis is

that the added attribute will potentially enhance the model's performance while requiring similar training epochs. The potential issue with this is that a number of graph candidates' underlying structures may be very similar, resulting in very similar graph-based features. Therefore, a technique known as *Principle Component Analysis (PCA)* is used to address this issue.

Principle Component Analysis

Principal Component Analysis (PCA) is a popular technique for reducing the dimensionality of data, particularly in Exploratory Data Analysis. A linear transformation technique projects the data onto a lower-dimensional space while retaining the most important information in the original data [93]. One of the key components of PCA is the Singular Value Decomposition (SVD) method. SVD is a mathematical technique that decomposes a matrix into three matrices. The first matrix captures the most important features of the original data, while the remaining two matrices represent the transformation from the original data to the reduced-dimensional space [94, 95].

Before applying SVD, it is essential to standardize the data by subtracting the mean of each feature from the corresponding feature without altering the scale of the feature variances. This step ensures that SVD captures the data's intrinsic structure, preparing it for more efficient analysis and interpretation. The scikit-learn Python library efficiently implements PCA and SVD [96]. This library makes it easy to implement these methods and provides additional utilities for data preprocessing, model selection, and evaluation.

Since the new features to be generated were at the node level, four node-level features were chosen: *Harmonic Centrality*, *Betweenness Centrality*, *Eigenvector Centrality*. and *Spectral Radius*. These were chosen because they required little computational expense; generating these values for each node for each graph in the entire dataset did not take long.

- **Harmonic Centrality** This is a measure of the importance of a node in a graph. It is calculated as the sum of the reciprocals of the shortest path distances from every other node to a

specific node u [97]. Mathematically, harmonic centrality is represented as:

$$C(u) = \sum_{v \neq u} \frac{1}{d(v,u)} \tag{3.3}$$

where $C(u)$ is the harmonic centrality of node u, and $d(v,u)$ is the shortest path distance between nodes v and u. Harmonic centrality is a useful metric for identifying important nodes in a graph. It takes into account the number of nodes connected to a specific node and the length of the paths between those nodes. The higher the harmonic centrality of a node, the more important it is in the graph. The harmonic centrality measure has been used in several applications, including social network analysis, community detection, and recommendation systems. It is a powerful tool for understanding the structure of complex networks and identifying key nodes that play important roles in them.

- **Betweenness Centrality** Quantifies the extent to which a node lies on the shortest paths between other node pairs. It is defined as:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{3.4}$$

where V denotes the set of nodes, $\sigma(s,t)$ is the total number of shortest paths from node s to node t, and $\sigma(s,t|v)$ counts those paths passing through node v, excluding direct paths between s and t [98].

- **Eigenvector Centrality** Assesses a node's influence based on the centrality of its neighbors, implying that a node is considered important if it is connected to other significant nodes. The centrality for node v is the i-th normalized element of vector $\mathbf{v}$, determined by:

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v} \tag{3.5}$$

Here, $\mathbf{A}$ is the adjacency matrix, and λ represents the largest eigenvalue [99].

- **Spectral Radius** Is the maximum absolute value among the eigenvalues ($|\lambda_1|, \ldots, |\lambda_n|$) of a graph's adjacency matrix:

$$\rho(\mathbf{A}) = \max\{|\lambda_1|, \ldots, |\lambda_n|\} \tag{3.6}$$

  The spectral radius provides insights into the graph's connectivity and robustness, indicating that a high value suggests a well-connected structure [100].

After computing the four additional node features in the second case study presented in Chapter 5, they were input into the PCA model to produce new graph-level features with a reduced dimensionality of two. In Figure 3.2, you can see the data plotted for both the "simulate" and the "compile" classes, where you can clearly see two distinct groupings within the data. Each axis represents a column from the output of the PCA analysis. The left-hand plot is the *simulate* classes where a 1 indicates the graph will simulate and a 0 means it will not simulate. Observing closely, you notice that the *simulatable* data is towards the top half of both the x and y axes. Similarly, the right-hand plot is the *compile* classes. The difference between simulate and compile is discussed further in Chapter 5 in Case Study 2.

In summary, PCA and SVD are powerful techniques for dimensionality reduction in data analysis. These methods reduce the dimensionality of the data, allowing for a more efficient analysis of large datasets and helping to uncover important relationships among variables in the data. A Python code sample can be found in Section A.2.

3.3 Model Architecture

In this study, the model comprises five layers: namely, three graph convolutional layers [101], a mean pooling layer, and a linear layer for the ultimate output. To achieve the objective of graph classification, the model follows a three-step process: 1) employs the convolutional layers to encode individual nodes by exchanging messages, 2) aggregates the node embeddings to form a graph representation, and 3) utilizes the graph representation to train the classification layer.

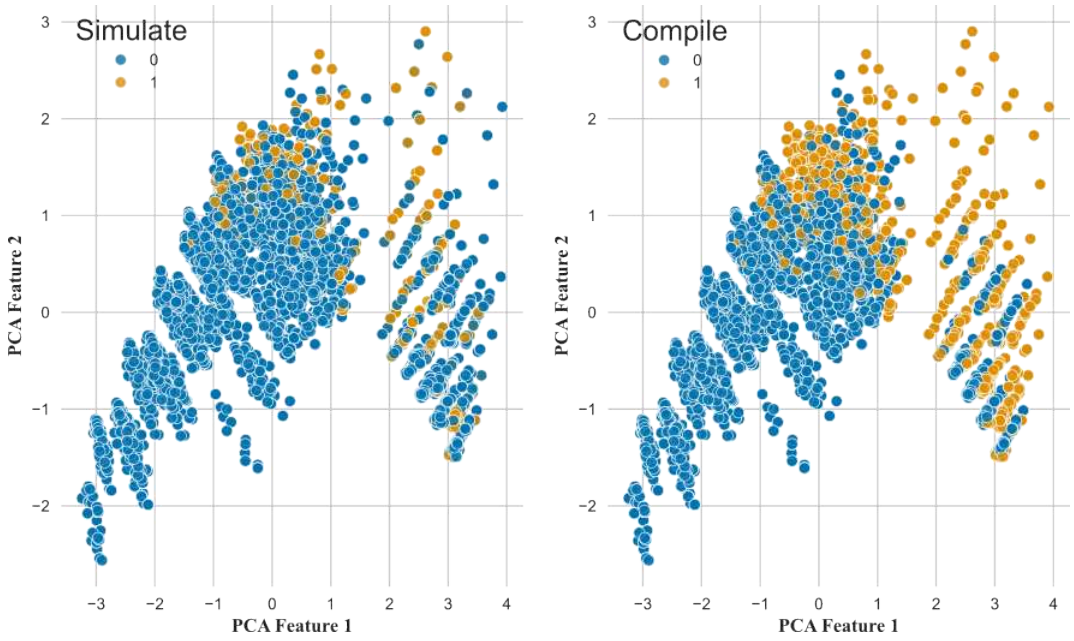


Figure 3.2: The Principle Component Analysis features displayed for both the simulate and compile classifications.

3.3.1 Graph Convolutional Layer

The Graph Convolutional Layer (GCN) operates to derive the output features $\mathbf{X}'$ following this formula:

$$\mathbf{X}'_i = \mathbf{W}_1\mathbf{X}_i + \mathbf{W}_2 \sum_{j \in \mathrm{N}(i)} e_{j,i} \cdot \mathbf{X}_j \tag{3.7}$$

where $\mathbf{X}$ represents the input feature for each node and $e_{j,i}$ denotes the edge weight from source node j to target node i [101]. The weights $(\mathbf{W}_1, \mathbf{W}_2)$ adapt during training iterations.

Each GCN layer also employs the Rectified Linear Unit (ReLU) activation function $f(x) = \max(0, x)$, which returns 0 for any negative input and returns the input value for positive inputs. Finally, the result from Equation (3.7) is passed through the activation function:

$$\mathbf{X}_{i+1} = f(\mathbf{X}_i) \tag{3.8}$$

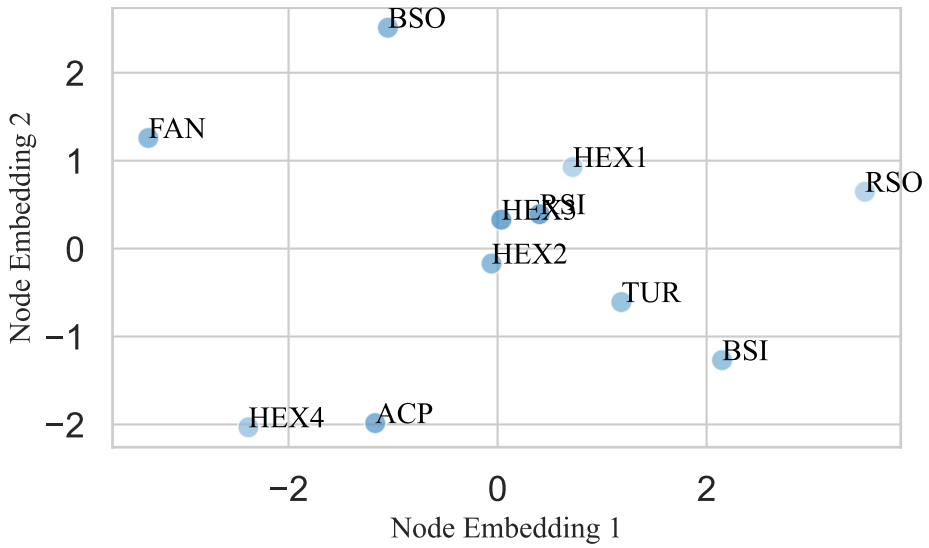


Figure 3.3: Computed node embeddings for the ACM graph in Figure 2.6b.

This step assists in obtaining localized node embeddings. Node embeddings involve translating the nodes or vertices of a graph into an N-dimensional numeric space, resulting in numerical vectors that encapsulate the graph's structure. This is exemplified through the node embeddings obtained for the ACM graph shown in Figure 2.6b in the second case study and depicted in Figure 3.3. In this illustration, it is evident that nodes HEX1, HEX2, and HEX3 (which are the top three HEX nodes in Figure 2.6b, respectively) along with RSI, have been identified as similar and, consequently, are clustered together.

3.3.2 Global Mean Pooling Layer

The subsequent layer is the Global Mean Pooling layer. This layer processes the final output from the GCN layers. It produces an output vector $\mathbf{r}$ by computing the mean of the node features $\mathbf{X}$, encompassing node embeddings for the entire graph, spanning all nodes within the graph N. This procedure generates a comprehensive graph embedding:

$$\mathbf{r}_i = \frac{1}{N_i} \sum_{n=1}^{N_i} \mathbf{X}_n \tag{3.9}$$

3.3.3 Linear Layer

The last layer in the model is the classification layer. It receives the output from the mean pooling layer as its input and carries out a linear transformation. Nevertheless, just prior to this layer, a "dropout" operation is employed, which stochastically sets certain elements of the input tensor to zero with a specified probability, utilizing samples from a Bernoulli distribution, as introduced in Ref. [102].

$$\mathbf{y} = \mathbf{r}\mathbf{A}^T + \mathbf{b} \tag{3.10}$$

3.4 Hyperparameters and Options

There are several items that can be tuned in the selected model architecture so that the model trains well on the type of data provided. For more information on hyperparameters and tuning, please refer to the following sources [103–105].

Learning Rate

In machine learning, the Learning Rate is a hyperparameter that plays a crucial role in the training process. It determines the size of the step response to the estimated error as the weights are updated with each epoch and is considered one of the more critical hyperparameters to tune [104].

The learning rate is a positive scalar value that typically ranges from 0.0 to 1.0. The choice of learning rate is essential for the success of the training process. When the learning rate is too small, the model will take too long to converge, and the training process will be slow. On the other hand, when the learning rate is too large, the model may overshoot the optimal weights and fail to converge, leading to an unstable training process. Therefore, it is essential to choose an appropriate learning rate that balances the training time and the stability of the training process. In general, a learning rate between 0.001 and 0.1 is a good starting point. However, the optimal learning rate value depends on various factors such as dataset size, model architecture, and mini-batching.

The case studies mentioned begin with a learning rate of 0.001 in conjunction with a *learning rate scheduler*. A learning rate scheduler adjusts the learning rate to optimize gradient descent. However, it is always essential to experiment with different learning rate values to find the optimal value for a given problem.

Number of Epochs

An epoch is a critical concept in machine learning, referring to the number of times the entire training dataset is passed through the model during training [106]. The model evaluates the entire dataset during each epoch, and the weights are updated after each iteration. In machine learning algorithms, the number of epochs is another hyperparameter, which allows the model to train until the error is minimized. The number of epochs can range from hundreds to thousands, depending on the complexity of the model and the dataset.

The number of epochs is a delicate balance between underfitting and overfitting the data. When the model is underfitting, it means that it is not learning enough from the data. Conversely, when the model is overfitting, it means that it is learning too much from the data, memorizing the training datasets, and losing the ability to generalize to new data.

In the two case studies, different experiments explore the effects of varying the number of epochs on the models' performance. The experiments aim to find the optimal number of epochs for training the model, considering the dataset's complexity and architecture. It is important to note that the optimal number of epochs may vary based on the task and the dataset. Therefore, it is essential to experiment with different values of epochs to find the one that best fits the specific problem.

Optimization Algorithm

Optimization algorithms play a crucial role in various fields, from engineering to finance. These algorithms are used to find the optimal input parameters of a function that result in the minimum or maximum of the function output [106].

One of the popular optimization algorithms is the Adaptive Movement Estimation (ADAM) algorithm, which is a stochastic optimization approach [72]. The algorithm is designed to compute individual learning rates for different parameters of the gradients' first and second moments. In simpler terms, the ADAM algorithm adjusts the learning rate of each weight in the neural network based on the gradient of the loss function to speed up convergence. The ADAM algorithm has several advantages over other optimization algorithms, such as faster convergence and a lower memory requirement. It has been widely used in deep learning models, including image classification, language translation, and speech recognition. In our model, we chose to use the ADAM algorithm as it provides better results than other optimization algorithms. The ADAM algorithm was applied to optimize the weights and biases of the neural network in our model. We conducted experiments to compare its performance with other optimization algorithms, and the results showed that it outperformed the others.

In conclusion, optimization algorithms are essential in various fields. The ADAM algorithm is a powerful optimization approach for optimizing the weights and biases of neural networks. We can achieve faster convergence and better results in our models by utilizing the ADAM algorithm.

Loss Function

A loss function is an essential part of training a machine learning model. It is a metric that calculates the difference between the expected output of the model and its current output. This difference is commonly known as the "loss" or "cost" of the model. The goal of training is to minimize this loss function.

There are different types of loss functions, and the choice depends on the problem being solved. In this model, we use the cross-entropy loss function, which is commonly used in classification problems. This function measures the difference between two probability distributions for a given variable. The cross-entropy loss function is particularly useful for classification problems because it quantifies the difference between the predicted probability distribution and the actual probability

distribution of the target variable. It is often used in problems where the output is a probability distribution over multiple classes, such as image classification and natural language processing.

The choice of loss function is crucial in training a machine learning model. If the loss function is not appropriate for the problem, the model may not converge to a good solution. Therefore, choosing a loss function appropriate for the problem being solved is important.

Batch Size

When training a deep-learning model, batch size is one important hyperparameter to consider. The batch size determines the number of data samples processed before updating the model weights. This technique is called mini-batching and can significantly improve the efficiency of the training process.

Rather than processing the data points one by one or all at once, mini-batching groups a set of data points of intermediate size. This approach has several advantages. First, it allows the model to scale better to large amounts of data, as it reduces the memory requirements of the training process. Second, it can speed up the training process by enabling vectorized operations, which can be more efficient than processing individual data points. Moreover, mini-batching can help prevent overfitting, as it introduces additional randomness into the training process. By randomly shuffling the data points and selecting batches at each iteration, mini-batching can reduce the risk of the model memorizing the training data instead of learning the underlying patterns.

To ensure our model's effectiveness, it is crucial to fine-tune its hyperparameters and rigorously evaluate its performance using appropriate metrics. In the next section, we will delve into the various performance metrics used to assess the accuracy and robustness of the model.

3.5 Model Performance Metrics

After a model has completed training, a *test dataset* was used to determine how well the model performs. In the test set, we know the correct values for the classification problem at hand. There-

fore, various Python modules make it easy to determine model performance using the metrics described in the following sections.

Confusion Matrix

A confusion matrix is a classification tool that displays the performance of a model. It is a two-dimensional matrix where the columns represent the instances as classified by the model, and the rows represent the instances as they actually are in the true dataset, as described in [107]. The confusion matrix is particularly useful for evaluating the performance of binary classifiers.

In the case of a binary classifier, the matrix is divided into four quadrants. The top-left quadrant shows the True Positives (T_P), which are the data points correctly identified as ones. The top-right quadrant represents the False Positives (F_P), where the data points are incorrectly labeled as ones when they are actually zeros. The bottom-left quadrant contains the False Negatives (F_N), indicating the points that were mistakenly classified as zeros but are truly ones. Finally, the bottom-right quadrant is for the True Negatives (T_N), which correctly identifies the data points as zeros.

Figure 3.4 provides an illustration of a confusion matrix. It is important to note that the confusion matrix is not limited to binary classification problems and can be extended to multi-class classification tasks as well.

The confusion matrix provides a quick and easy way to evaluate a classifier's accuracy. By examining the values in each quadrant, a range of metrics, such as accuracy, precision, recall, and F1 score, can be calculated, which can provide a more detailed understanding of the classifier's performance.

- **Accuracy** — Accuracy is an important metric used to evaluate the performance of predictive models. It measures how well a model is able to correctly classify instances into their respective categories. In other words, accuracy measures the percentage of correct predictions made by the model out of the total number of predictions. For example, if a model is trained to classify images of cats and dogs, and it predicts that 90 out of 100 images are cats, and

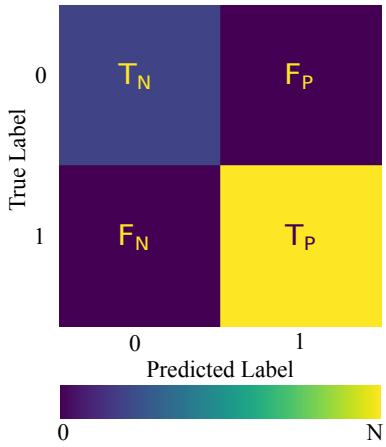


Figure 3.4: An example confusion matrix

they are indeed cats, then the accuracy of the model would be 90%, see Equation (3.11).

$$Accuracy = \frac{T_P + T_N}{N} \tag{3.11}$$

It is important to note that accuracy alone may not be sufficient to evaluate a model's performance. Other metrics, such as precision, recall, and F1-score, may be more appropriate in some cases, depending on the problem being solved. Overall, accuracy is a useful measure of a model's performance and is commonly used in various fields such as machine learning, data science, and artificial intelligence.

- **Precision** — The precision, also known as Positive Predictive Value, is a crucial metric that provides us with the proportion of positive classifications that were actually correct. It is particularly helpful when dealing with data that has a class imbalance, where one class has significantly more instances than the other. Training your model on such a dataset may classify all data as the most frequent class, leading to misleadingly high accuracy. Precision is a class-specific metric defined as follows:

$$Precision = \frac{T_P}{(T_P + F_P)} \tag{3.12}$$

where T_P is the number of true positives and F_P is the number of false positives. In other words, Precision measures the percentage of positive predictions that were actually correct.

This same metric can also be applied to the negative value, and it is referred to as Negative Predictive Value (NPV). However, this paper will focus on Precision since our case studies aim to find the best architecture. In conclusion, Precision is a crucial metric that helps us to determine the accuracy of positive predictions. It is particularly useful when dealing with datasets that have a class imbalance. Therefore, precision and other metrics should be considered when evaluating a machine-learning model's performance.

- **Recall** — This metric, also known as the True Positive Rate or Sensitivity, is a measure of the accuracy with which a model or system can identify actual positive classifications. In other words, it determines the proportion of true positive cases that were correctly identified out of all the positive cases present in the dataset. A high true positive rate indicates that the model effectively detects positive cases and minimizes false negatives, which are cases where a positive classification was missed.

$$\textit{Recall} = \frac{T_P}{T_P + F_N} \tag{3.13}$$

- **F1 Score** — The F1 Score is a widely used metric in machine learning to evaluate the performance of a model. It is a measure of a model's accuracy, combining both Precision and Recall into one score. Precision and Recall are two important metrics used to evaluate classification models' performance. Precision measures the accuracy of the model's positive predictions, while Recall measures the number of true positives the model correctly identifies. The F1 Score is calculated by taking the harmonic mean between Precision and Recall. The harmonic mean places more weight on lower values, meaning that a model with both high Precision and high Recall will have a higher F1 Score than a model with high Precision

but low Recall or vice versa.

$$F1\ Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3.14}$$

A high F1 Score indicates that the model has both high precision and high recall, which means that it is performing well in correctly identifying positive cases while minimizing false positives and false negatives. In summary, the F1 Score is a useful metric for evaluating the performance of classification models. It provides a single score that takes into account both Precision and Recall, making it a reliable measure of a model's accuracy.

- **Matthews Correlation Coefficient (MCC)** — This metric is widely used as an evaluation metric for classification models. It is a performance metric that measures the quality of binary (two-class) classifications. MCC is particularly useful when the classes are imbalanced in size. It also provides a balanced measure of the quality of the classification, irrespective of the distribution of the classes.

$$MCC = \frac{T_P \cdot T_N - F_P \cdot F_N}{\sqrt{(T_P + F_P)(T_P + F_N)(T_N + F_P)(T_N + F_N)}} \tag{3.15}$$

The MCC ranges from -1 to 1. A score of 1 indicates that the model can make predictions perfectly, a score of -1 means that every prediction was incorrect, and a score of 0 indicates that the model is just as good as random chance. A high MCC score is a sign of a good classification model. MCC produces a high score if the model obtained good results in all boxes of the confusion matrix. This means high T_P and T_N values and low F_P and F_N values.

- **Total Set Accuracy** — When the datasets are broken down into their respective subsets,

$$(\mathcal{G}_{known}, \mathcal{G}_{unknown})$$

we still want to know how well the predictions are when compared to the $\mathcal{G}_{all}$ if it is available. Therefore, we define *Total Set Accuracy* as:

$$\textit{Total Set Accuracy} = \frac{T_P^{(u)} + T_N^{(u)} + T_P^{(k)} + T_N^{(k)}}{N_{all}} \tag{3.16}$$

where $(T_P^{(u)}, T_N^{(u)})$ is determined on $\mathcal{G}_{unknown}$ using the trained model, and $(T_P^{(k)}, T_N^{(k)})$ is determined using $\mathcal{G}_{known}$, which might not be perfectly correct due to the median difference.If there is no misclassification based on the different medians, then $T_P^{(k)} + T_N^{(k)} = N_{known}$. This metric represents a scenario where a designer classifies graphs using both known and unknown data.

In summary, the metrics discussed provide essential insights into model performance, ensuring that the evaluation is comprehensive and diverse. Understanding these metrics is crucial for assessing the effectiveness of our models and guiding improvements.

Building on this foundation, we will now discuss the core methodology of Iterative Classification. This approach plays a pivotal role in the IC-GSBD framework, leveraging the metrics we discussed to refine and enhance the model's predictive capabilities through successive iterations.

3.6 Iterative Classification

Originally defined in Ref. [51], Iterative Classification (IC) has proven to be an effective tool for the down-selection process, ensuring narrowing down to the top-performing solutions based on a desired metric. In particular, when we seek binary classification for best graph performance from Section 3.1.1, classifying based on median performance (once) only reduces the set of $\mathcal{G}_{unknown}$ by approximately half. This outcome can still result in too many graphs to evaluate, so we might consider iteratively building more focused GDL models based on better threshold values for the "good" performance classification. First, we will discuss how the datasets are broken down.

3.6.1 Dataset Preparation

Based on the Type 1 problem classification considered here from Section 2.2.2, we will consider the case when only some of the performance values $J(G_i)$ for $G_i \subset \mathcal{G}$ are known. This will divide the graphs into two sets as follows:

$$\mathcal{G} \equiv \mathcal{G}_{all} = \mathcal{G}_{known} \cup \mathcal{G}_{unknown} \tag{3.17}$$

where $\mathcal{G}_{known}$ is the set of graphs with known values for $J(G_i)$ and $\mathcal{G}_{unknown}$ represents graphs with unknown $J(G_i)$ values (and this is what the GDL model is for). We will denote the sizes of the sets as $|\mathcal{G}_{all}| = N_{all}$, $|\mathcal{G}_{known}| = N_{known}$, and $|\mathcal{G}_{unknown}| = N_{unknown}$. They also satisfy:

$$N_{all} = N_{known} + N_{unknown} \tag{3.18}$$

We also note that this is a bit different than other types of datasets used in machine learning as there is a known, finite amount of potential inputs. The goal here is to develop accurate models where $N_{known} \ll N_{unknown}$.

As is typical in machine learning, we will create two subsets from $\mathcal{G}_{known}$:

$$\mathcal{G}_{known} = \mathcal{G}_{training} \cup \mathcal{G}_{validation} \tag{3.19}$$

where $\mathcal{G}_{training}$ is the training dataset and $\mathcal{G}_{validation}$ is the validation dataset. The model fits using $\mathcal{G}_{training}$, and the fitted model is used to predict the responses for the observations in $\mathcal{G}_{validation}$ [105]. For example, after each iteration, the model will adjust its weights accordingly and test them on the validation set, which can help understand model performance and adjust options as necessary. The unknown set is used when the model is finalized, and you want to now test it on data that the model has not seen.

As shown in Figure 3.1, the binary classification of graphs in $\mathcal{G}_{known}$ is based on the median performance value (or potentially some other dividing line using the known data). Since we only

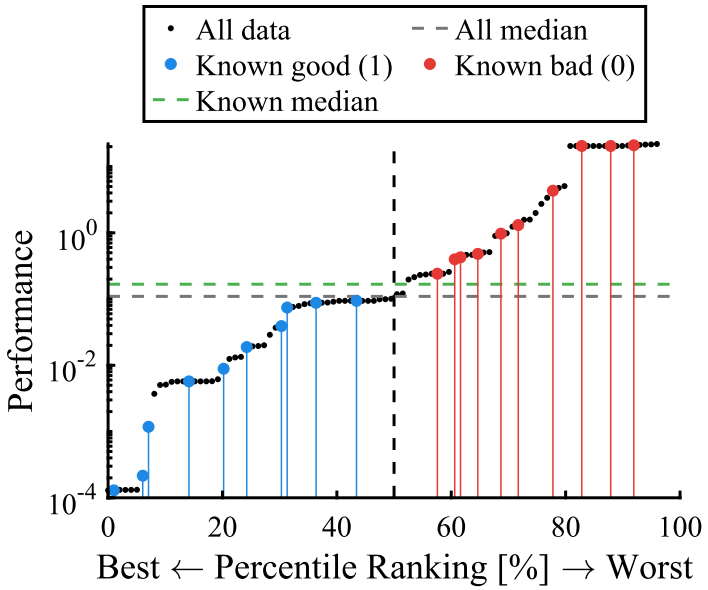


Figure 3.5: An Illustration of $\mathcal{G}_{all}$ and $\mathcal{G}_{known}$ for 100 graphs, the performance-based classification of $\mathcal{G}_{known}$, and the potential difference between the medians from the circuit dataset.

know $J(G_i)$ in $\mathcal{G}_{known}$ rather than $\mathcal{G}_{all}$, the median performance value used for the initial labeling might be different than the true value if we had all the performance values for $\mathcal{G}_{all}$. However, this error will become small as the relative size of $\mathcal{G}_{known}$ increases. The concepts in this section are illustrated in Figure 3.5. Here we have 100 graphs in $\mathcal{G}_{all}$ and 20 randomly selected graphs for $\mathcal{G}_{known}$. The true median and $\mathcal{G}_{known}$-based median do differ, and two graphs are between these lines.

3.6.2 The IC Algorithm

The iterative classification approach is outlined in Algorithm 1. The initial step and key variable to this approach is determining the initial value for $\mathcal{G}^k_{known}$. There is not a predefined approach to determining this value, but there are factors that go into it, such as the size of $\mathcal{G}_{all}$, the time it takes to evaluate $\mathcal{G}^k_{known}$, and the variation in the graph topologies and features. If the graph features (graph level, node, or edge) and graph topology vary widely, the model may need to be trained on a larger number of graphs to see all the different variations. In the first case study in Chapter 4, there was little variation in features and topology, therefore only requiring a small amount of the dataset for the model to train on.

Algorithm 1 Iterative Classification for Graph-Set Based Design Algorithm

Input: $|\mathcal{G}_{known}|_{min}$, n, $\mathcal{G}_{all}$
$k \leftarrow 1$ ▷ Initialize iteration counter
Determine N^k_{known}
Randomly select N_{known} graphs from $\mathcal{G}_{all}$
Evaluate $\mathcal{G}_{known}$
Classify $\mathcal{G}^k_{\text{known}}$ into "Known 1" and "Known 0" based on $\bar{J}(\mathcal{G}^k_{known})$
while $k \leq n$ **do** ▷ Iterate until specified limit
    Train GDL model m^k using $\mathcal{G}^k_{known}$ ▷ Train the model
    $m^k(\mathcal{G}^k_{unknown})$ ▷ Model predicts on $\mathcal{G}_{unknown}$
    Form sets "Predicted 1" and "Predicted 0" from these predictions
    Discard "Predicted 0" and "Known 0" Classes
    Set $\mathcal{G}^{k+1}_{known} \leftarrow$ "Known 1"
    if $N^{k+1}_{known} < |\mathcal{G}_{known}|_{min}$ **then** ▷ Check if known dataset is below the minimum size
        Supplement $\mathcal{G}^{k+1}_{known}$ with graphs from "Predicted 1" until $N^{k+1}_{known} = |\mathcal{G}_{known}|_{min}$
        Divide $\mathcal{G}^{k+1}_{known}$ into "Known 1" and "Known 0" based on $\bar{J}(\mathcal{G}^k_{known})$
        Set $\mathcal{G}^{k+1}_{unknown} \leftarrow$ "Predicted 1"
    else
        Divide $\mathcal{G}^{k+1}_{known}$ into "Known 1" and "Known 0" based on $\bar{J}(\mathcal{G}^{k+1}_{known})$
        Set $\mathcal{G}^{k+1}_{unknown} \leftarrow$ "Predicted 1"
    end if
    $k \leftarrow k + 1$ ▷ Increment iteration counter
end while

The next important value is the minimum value for the size of $\mathcal{G}^{n-1}_{known}$, where n is the number of chosen iterations. As iterations continue, and we narrow down to the top graphs in the set, $\mathcal{G}_{known}$ is going to continue to shrink as we remove graphs that were predicted as "bad". But, a minimum number of graphs is required for the model to train and be effective. Therefore, if we set a minimum value for this set, $|\mathcal{G}_{known}|_{min}$, we can supplement it with predicted 1s from the previously completed iteration.

When training begins for the model, it trains on the initial $\mathcal{G}^1_{known}$. After training is complete, the model m^1 is used to predict the classes for $\mathcal{G}^1_{unknown}$. Once the $\mathcal{G}^1_{unknown}$ is divided into "Predicted 1s" and "Predicted 0s," the predicted zeros are determined to not be feasible solutions and are therefore discarded. The "Known 0" class from $\mathcal{G}^1_{known}$ are also determined to be infeasible and are discarded as well.

The $\mathcal{G}^1_{known}$ "Known 1" class becomes $\mathcal{G}^2_{known}$ and its size is denoted as N^2_{known}. With each iteration, the size of $\mathcal{G}^k_{known}$ is reduced significantly and at this point, if $N^2_{known} < |\mathcal{G}_{known}|_{min}$, $\mathcal{G}^2_{known}$ is supplemented with predicted ones. Lastly, we divide $\mathcal{G}^2_{known}$ into its respective "Known 1" and "Known 0" classes, the "Predicted 1" class becomes the $\mathcal{G}^2_{unknown}$, and the process starts again by training the next model on the new known set, $m^2(\mathcal{G}^2_{known})$. During this process, the good/bad classification threshold for the GDL model m^k adjusts accordingly, thus narrowing down to the top-performing graphs.

As you will see in Chapter 4 and Chapter 5, the IC process is an efficient and effective tool for determining the best possible solutions in large datasets. Although it does not determine the *top* performing graph, it gives the designer a much smaller set that can be further evaluated, giving stakeholders more than one option to choose from based on pre-determined requirements. In the next section, we explore the specific contributions of Geometric Deep Learning to enhancing Set-Based Design methodologies.

3.7 Role of Geometric Deep Learning and Iterative Classification in Graph Set-Based Design

The Graph-Set-Based Design (GSBD) exemplifies the innovative application of Set-Based Design (SBD) principles in conjunction with graph-based engineering designs and Geometric Deep Learning (GDL). This integration leverages the SBD framework to significantly enhance the design process by broadening the exploration of design alternatives. Graph-based design sets represent various solutions, embracing the SBD principle of extensive exploration. By harnessing GDL's capabilities, IC-GSBD enriches the SBD framework, enabling efficient evaluation and selection of design alternatives. GDL's proficiency in processing graph representations expedites the assessment of each design's feasibility and performance, streamlining the refinement process [69]. IC-GSBD enhances decision-making with computational efficiency, embodying systematic and iterative exploration central to SBD, and reducing the number of required design iterations [51].

The IC-GSBD methodological framework utilizes the SBD approach to categorize and refine graph-based designs. As illustrated in Figure 2.1, the process begins with defining requirements and generating design alternatives, which are then manually evaluated and narrowed down to determine the optimal design solution. In Figure 3.6, GDL and iterative classification are integrated into the foundational SBD workflow.

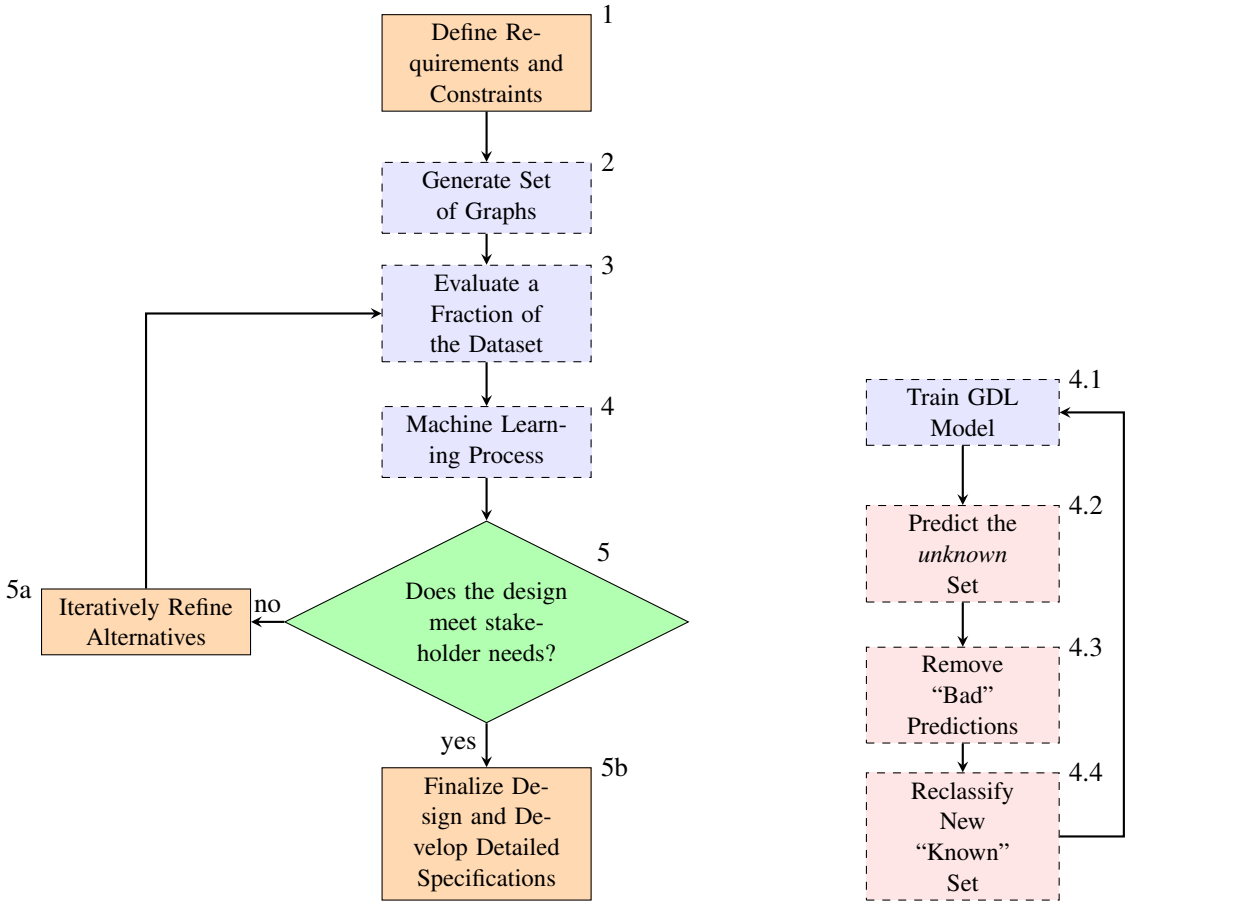
Taking a closer look at Figure 3.6a, we can see that Step 1 in both GSBD and SBD are the same; both frameworks have to define the requirements and constraints. Steps 2–4 are what differentiates GSBD from SBD. In Step 2, we generate the set of graphs. This could be done by various means, but the method used in the case studies is graph enumeration described in Section 2.2.2. Step 3 is also very similar to SBDs' as you are required to analyze the design alternatives. What makes this step more unique in GSBD is that a designer only has to evaluate a small fraction of the dataset, this is the set that is used in the machine learning process (Step 4). In Figure 3.6b, you can see an exploded view of the machine learning process where Steps 4.2-4.4 are the processes that contain automation within the IC method.

One significant advantage of this revised workflow, depicted in Figure 3.6a, is that it does not introduce additional steps to the existing process. Furthermore, Figure 3.6b presents an exploded view of the machine learning process, highlighting the incorporation of automated processes that efficiently refine the alternatives.

This enhanced methodology fits seamlessly within the traditional SBD framework, enhancing it by streamlining the management of computational complexity associated with expansive design explorations. It provides a robust mechanism for evaluating and predicting the performance and feasibility of a broad range of design possibilities, represented as graphs, with remarkable efficiency and effectiveness.

3.8 Conclusion

In this chapter, we have provided a detailed overview of the methods used in the IC-GSBD methodological framework. We began by discussing the role of graph classification, highlighting



(a) The basic workflow for Graph-Set-Based Design.

(b) An exploded view of the machine learning process for the iterative classification process.

Figure 3.6: The IC-GSBD workflow where steps 2–4 indicate the changes to the former SBD process shown in Figure 2.1, and steps 4.2–4.4 indicates automated processes in iterative classification.

the differences between binary classification and multi-label classification and how they are used. Both forms of graph classification played important roles in the case studies discussed in the next chapter.

We have also demonstrated the effectiveness of feature engineering by showing that additional features for the model can be obtained without requiring all the information from the system. We discussed how to obtain node-level and graph-level features and how principle component analysis can be used to condense the dimensionality of multiple node features.

The model architecture was presented, with a description of each layer's purpose. We explained node embeddings and their impact on a graph during model training. Additionally, we detailed each

hyperparameter within the model that can be fine-tuned to improve performance. We also outlined iterative classification and its associated algorithm and discussed how SBD can benefit from GDL and graph-based systems.

In the upcoming chapters, we will introduce two case studies. The first case study involves the down-selection of analog circuits, where we demonstrate how iterative classification can narrow down 43,000+ graphs to the top-performing circuits. The second case study applies similar techniques to a different set of graphs, consisting of 32,600+ directed graphs, as opposed to the undirected graphs in the first case study. These two case studies will demonstrate the versatility of IC-GSBD.

Chapter 4

Case Study 1: Down-Selection of Analog Electric Circuits[3]

> "Design is not just what it looks like and feels like. Design is how it works."
>
> –*Steve Jobs*

In this chapter, we explore the capabilities of Iterative Classification for Graph-Set-Based Design (IC-GSBD) in graph-based engineering design problems by demonstrating the down-selection of analog circuits, utilizing the results of a frequency response matching study [17]. Analog circuit design is a fundamental aspect of electrical engineering, requiring precise optimization to meet specific performance criteria. Traditional design methods often rely on iterative processes that can be time-consuming and computationally intensive. Integrating Geometric Deep Learning (GDL) with Set-Based Design (SBD) offers a novel approach to streamlining this process.

We will explore how the IC-GSBD methodological framework can effectively narrow down a vast set of potential circuit designs to a manageable subset of high-performing solutions. This case study demonstrates the practical application of IC-GSBD, emphasizing the benefits of reduced computational overhead and improved decision-making efficiency. The methods and results presented here provide a compelling argument for the adoption of IC-GSBD in complex engineering design tasks, showcasing its potential to revolutionize traditional practices and enhance innovation in analog circuit design.

[3]Parts of this chapter are based on Ref. [51]

4.1 Background

We begin by discussing the graph and how the dataset was generated through enumeration, followed by the metric that determined the graphs' performance value. Then, we explore the various experiments conducted beginning with establishing a baseline, followed by feature engineering. We then determine the appropriate number of epochs and the optimal value of N_{known}. Lastly, after analyzing the results from the previous experiments, we use that information for the iterative classification process.

4.1.1 Enumeration of Circuit Graphs

Automating the design synthesis of analog circuits has long been a problem of interest with numerous attempts [108, 109]. Still, these approaches can often fail to meet the needs when encountering unfamiliar and complex design problems. Even with the addition of heuristics [110] and knowledge bases [111], determining global solutions, sets of alternatives, and patterns is not straightforward, which leads us to consider enumeration.

As discussed in Section 2.2.2, we will consider the graph enumeration of electric circuits from Ref. [17]. In particular, the set $\mathcal{G}$ with 43,249 undirected graphs includes all topologies with up to 6 impedance subcircuits with RC components and a required connection to the ground [17]. Two examples are shown in Fig. 4.1 with RC values optimally selected for the design problem presented in the next section, along with the distribution of performance values $J(G_i)$.

Many researchers have utilized the enumeration technique of circuit problems [29,112,113] and in other problem areas where the data can be represented as graphs and other enumerable objects [114–116]. As mentioned, each circuit here is represented by a node-labeled graph, meaning every node has an associated label representing some circuit concept (and this component label is the single basic node-level feature of the dataset). The size of each graph varied between 6 and 20 nodes (which is relatively small for GDL), all of which had guaranteed different topologies (i.e., no labeled graph isomorphisms from Section 2.2.2). Finally, we will now describe a single graph-level feature, performance $J(G_i)$.

4.1.2 Frequency Response Matching Optimization

The performance value $J(G_i)$ here is the error between the desired frequency response and the one a selected circuit provides, based originally on the study in Ref. [108]. A circuit graph G_i does not have an intrinsic value for J; rather, it is a function of the *tunable values* for the RLC coefficients in the given graph and a selected *optimization problem* with an objective and constraints. From Ref. [17], we consider the "set 1" problem defined as:

$$\underset{\mathbf{z}_i=\{\mathbf{R}_i,\mathbf{C}_i\}}{\text{minimize:}} \quad J = \sum_k \left(\log|H_i(j\omega_k,\mathbf{z}_i)| - \log|F(j\omega_k)|\right)^2 \tag{4.1a}$$

$$\text{subject to:} \quad 10^{-2} \leq R_j \leq 10^{0} \quad \text{for all } R_j \text{ in } G_i \tag{4.1b}$$

$$10^{-2} \leq C_j \leq 10^{0} \quad \text{for all } C_j \text{ in } G_i \tag{4.1c}$$

$$\text{where:} \quad |F(j\omega)| = \sqrt{\frac{2\pi}{10\omega}} \quad 0.2 \leq \frac{\omega}{2\pi} \leq 5 \tag{4.1d}$$

where $|F(j\omega)|$ is the desired frequency response, $H_i(j\omega_k,\mathbf{z}_i)$ is the frequency response for circuit graph G_i, $\mathbf{z}_i$ is the collection of optimization variables for the resistors R and capacitors C in graph G_i, and ω is sampled at 500 logarithmically-spaced evaluation points.

Solving this nonlinear constrained (with simple bounds) least squares optimization problem can be expensive; the original study required over 8 hours to optimize each graph to determine $J(G_i)$. Therefore, reducing the computational costs to discern good and bad circuit graphs is desirable.

The performance values here have a large range, between 4×10^{-4} to 2×10^{2}, as illustrated in Fig. 4.1. Classification is done based on a sampled $\mathcal{G}_{known}$ using the median value of the known performance values, as discussed in Section 3.1.1. As we have the performance values for all graphs, this problem can serve an a good example to explore the potential effectiveness of GDL in these kinds of problems, so we can compare to classification using $\mathcal{G}_{all}$.

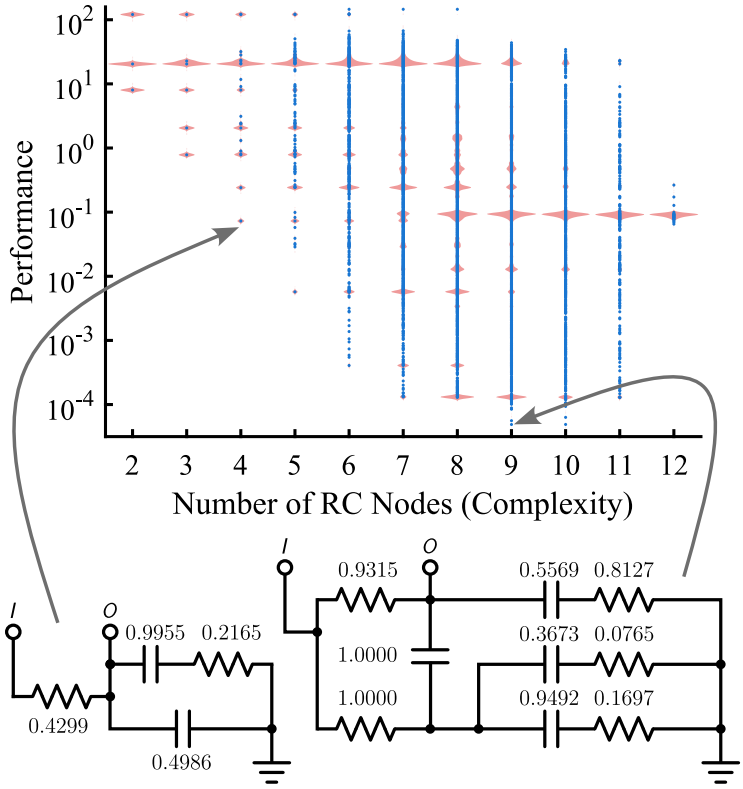


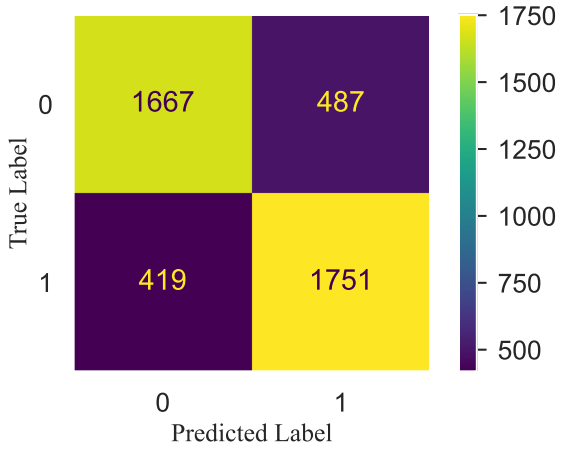
Figure 4.1: Summary of the number of RC nodes (a complexity metric) versus performance $J(G_i)$ for all 43,249 graphs with individual points and density in the frequency response matching optimization case study dataset.

4.2 Trials

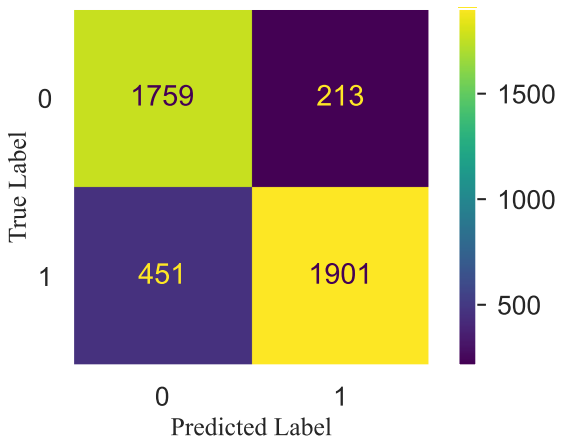
1: Establish a Baseline

In the first trial, we start by creating a baseline model for future comparisons. Here we consider a 72% (31,139 graphs) in the training set $\mathcal{G}_{training}$, 18% (7,784) in the validation set $\mathcal{G}_{validation}$, and the remaining 10% (4,324) in $\mathcal{G}_{unknown}$. These are by no means recommended distributions but rather a scenario with lots of data available for the GDL model that will be used to explore what is possible and what might be done to balance accuracy versus efficiency.

The CM for this trial is in Figure 4.2a using the 4,324 graphs in $\mathcal{G}_{unknown}$ that we have not used in any way to train the model (and in practice, you would not have this CM). Using Equation (3.11), we find a 79.1% accuracy, fairly close to the baseline models' final accuracy value for $\mathcal{G}_{known}$ of 79.5%. Next, using Equation (3.12), we calculate that the baseline model has a *Precision* of 77%.



(a) Confusion matrix for the *baseline* model predicting $\mathcal{G}_{unknown}$



(b) Confusion matrix for the *3-feature* model predicting $\mathcal{G}_{unknown}$

Figure 4.2: The baseline and feature engineering confusion matrices

Using Equation (3.13), we calculate the baseline models' *Recall* to be 79.9%, and lastly, using Equation (3.14) and Equation (3.15), we calculate the *F1 Score* to be 78.6% and *MCC* to be 58%, respectively.

2: Additional Graph-based Features

Here we consider adding additional graph-based features to $\mathbf{X}$ beyond the current vertex labels of R, C, G, etc. where the additional features should have a greater ability to explain the variance in the training data (known as feature engineering and feature selection). The theory is that these features will improve the model's performance for the same number of training epochs. The two features added here are eigenvector centrality and betweenness centrality, as discussed in Section 3.2.

Again, we point out that the computational expense of determining these additional features is quite low compared to calculated $J(G_i)$, so they add a relatively minor cost for each graph. With these two additional features, the new features matrix $\mathbf{X}$ for each individual graph has gone from $\mathbf{X} \in \mathbb{R}^{n\times 1}$ to $\mathbf{X} \in \mathbb{R}^{n\times 3}$. The CM for this trial is in Figure 4.2b. Here we have that by adding the two additional features, the model was able to achieve an accuracy of 85%, *Precision* of 89.9%, *Recall* of 80.8%, *F1 Score* of 85%, and an *MCC* of 70%. Therefore, the GDL model was able to

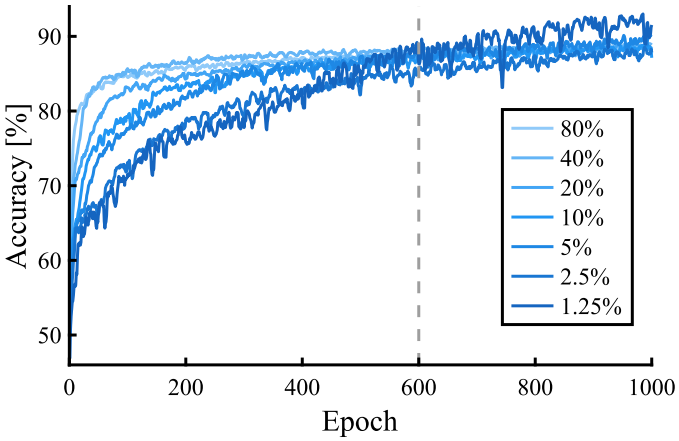


Figure 4.3: Accuracy of seven different models during training using different percentages of the dataset to determine the approximate number of epochs stopping point of 600.

make better predictions on the same $\mathcal{G}_{known}$ dataset as all metrics are the same or better. We will include all three features going forward.

3: Number of Epochs & Number of Known Graphs

The previous trials assumed a large number of epochs, but it is desirable to have insights into how many are actually required to effectively train the model to reduce computation costs. Here we will observe the training behavior up to 1,000 epochs and decide a limit for future trials.

Additionally, using 80% of the data for training, generally as was done in the previous trials, will not be desirable, especially when the cost of each $J(G_i)$ is quite high. Therefore, we will also explore in this trial different percentage values of $\mathcal{G}_{known}$, which represents fewer circuits that are sized using the expensive Eq. (4.1). The goal is to determine (for this particular dataset) a rule for the relative size of $\mathcal{G}_{known}$ to $\mathcal{G}_{all}$ that maintains the suitable accuracy while reducing the overall computational burden (both through model training time and time to generate $\mathcal{G}_{known}$). We will expect the model's performance to degrade as the amount of training data decreases.

Number of Epochs

Seven different models were training using different N_{known} spaced between 1.25% and 80%. Observing the results in Figure 4.3, there is a clear distinction around 600 epochs where many of

Table 4.1: The results for different values of N_{known} with all metrics computed with respect to the unknown graphs $\mathcal{G}_{unknown}$.

N_{known} [# graphs]	34,599	17,300	8,650	4,325	2,162	1,081	541	270	135
% of N_{all}	80%	40%	20%	10%	5%	2.5%	1.3%	0.6%	0.3%
Training Time [s]	4,294	2,239	1,257	720	464	334	269	243	229
Accuracy [%]	83.80	83.99	82.70	81.73	78.68	76.15	72.28	66.70	60.69
Precision [%]	84.10	88.37	80.29	79.11	77.18	76.51	70.97	70.54	67.70
Recall [%]	83.22	78.52	86.85	86.30	81.37	75.49	75.41	57.39	40.90
F1 Score [%]	83.66	83.15	83.44	82.55	79.22	75.99	73.13	63.29	51.00
MCC	0.68	0.68	0.66	0.64	0.57	0.52	0.45	0.34	0.23

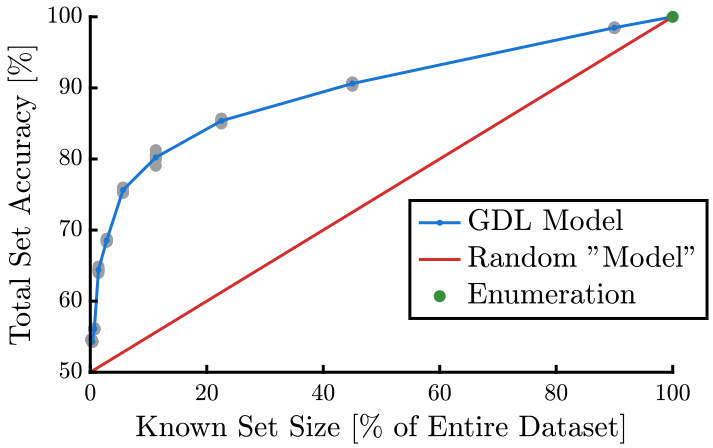


Figure 4.4: Total set accuracy scores averaged over five runs for different known set sizes N_{known}.

the accuracy metrics plateau. For some of the smaller training sets, accuracy continues to increase beyond this point, but it was observed that this was mostly overfitting to the small training dataset. Therefore, a 600 epoch limit is established and will generally result in models that are suitably trained without wasting additional iterations.

Number of Known Graphs

Using the same setup, we can explore trade-offs in the number of graphs (represented as a percentage here) included in $\mathcal{G}_{known}$. Reducing N_{known} has the consequence that $N_{unknown}$ increases, per Eq. (3.18), so more graphs will need their classifications predicted. Therefore, this trial aims to see what minimum amount of data is required for the models to retain a high level of accuracy.

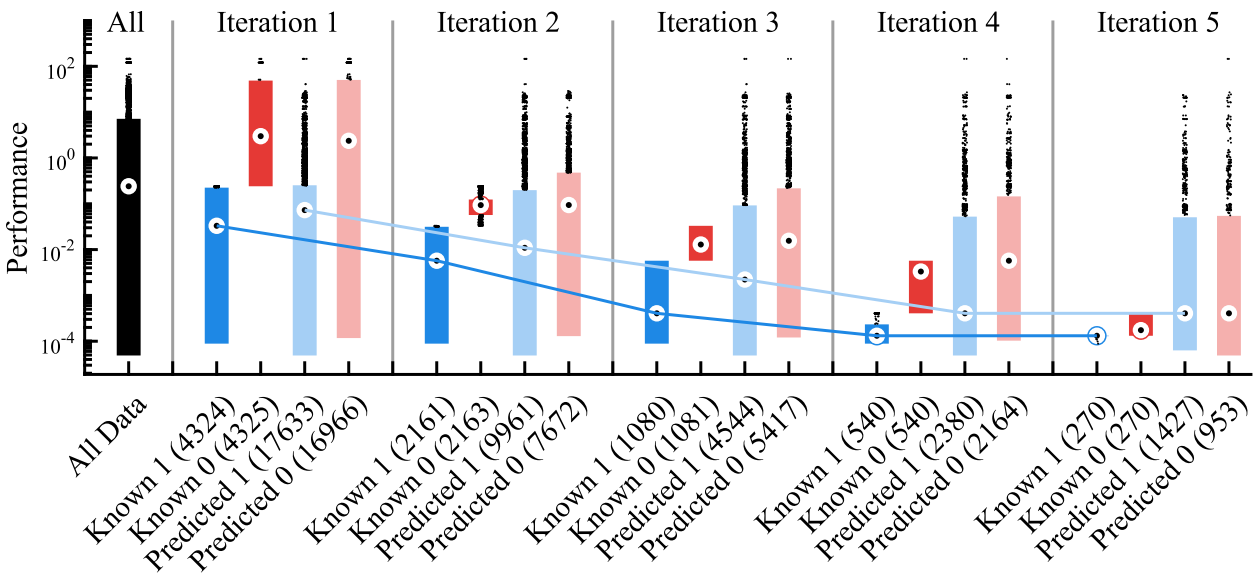


Figure 4.5: Five iterations of the approach described in Section 4.2 for retraining a GDL model based on successive performance minimizing-focused subsets of the graphs (with the numbers indicating the number of graphs in the respective set) along with the distribution of all the data points $J(\mathcal{G}_{all})$ for comparison.

The results are shown in Table 4.1 with all metrics computed using the unknown graphs $\mathcal{G}_{unknown}$. As we might expect, the larger N_{known} is, the higher the model's accuracy. Now, simply reducing the amount of data in half from 80% to 40%, we still achieve a high accuracy of ≈83% and similar *MCC* of 0.68. However, a key takeaway is the training time. The 80% model took 4,294 s (or about 72 min), while the 40% model took only 37 minutes; the training time in half by cutting the data in half. We should also consider the impact on computational cost required to construct $\mathcal{G}_{known}$; 40% will nominally take half as long to determine all required $J(G_i)$. Both of these factors are important in understanding what % of N_{all} is desirable when balancing total computational cost and model effectiveness (e.g., accuracy). Looking into the smaller N_{known} values, the results are also as expected: the smaller N_{known}, the lower many of the scores. At the limit of the runs, only 135 graphs (0.3%), the model was only able to achieve an accuracy of ≈60% with a much lower *MCC* of 0.23.

To better understand the trade-offs in using GDL models for the case study, we visualize the results of the metric *Total Set Accuracy* from Eq. (3.16) in Figure 4.4. As there are several stochastic elements to the approach, the mean of five different runs is shown with the different gray dots

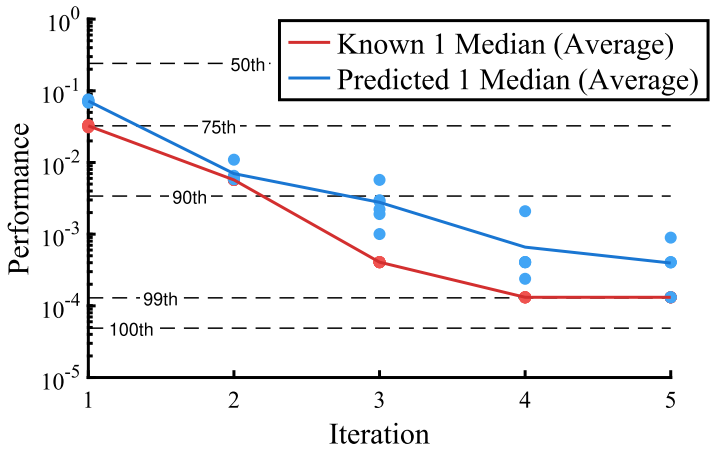


Figure 4.6: Median values of the "Known 1" and "Predicted 1" sets averaged over six runs using the iterative classification approach with different performance percentiles for $J(\mathcal{G}_{all})$ shown.

indicating the values for the individual runs. Overall, the variation is relatively smaller between runs. In addition to this curve, a theoretical random "model" is added where we assume that all known values are predicted correctly, and all others are randomly assigned a good/bad classification. Finally, when the known set size is 100% of the dataset, then we have enumerated all graphs, so the total set *Total Set Accuracy* is naturally 100%. Therefore, the IC should be compared with respect to these additional standards.

Overall, we see the GDL model greatly outperforms the random model between around 1%–40%, but the gap begins to close as more graphs are known and all approach the 100%/100% enumeration point as N_{known} increases. Therefore, a general recommendation might be to select N_{known} as 20% of N_{all}, but this value certainly can be problem-specific and depend on the designer's preferences to balance *Total Set Accuracy* vs. computational costs (which is generally proportional to N_{known}).

4: Iterative Classification

Even if with perfect classification, the approach outlined so far would only result in determining the top 50% performing graphs in $\mathcal{G}_{all}$. While this outcome can undoubtedly be useful in practice, it is generally desirable to narrow down the graphs further to the best-performing. However, there is no reason we only need to construct a single classification model.

Utilizing Algorithm 1, the results of this trial initialized with 20% of the data known are summarized in Figure 4.5. First, the leftmost box plot shows the distribution of $J(\mathcal{G}_{all})$. The next two box plots show the splitting of $\mathcal{G}_{known}$ based on the classification threshold at iteration 1. The final two box plots in iteration 1 are the predicted 1s and 0s based on the GDL model; note that, while there are certainly incorrectly identified graphs, the medians are quite separated and consistent with their "Known" counterparts. The subsequent box plots show the results for each iteration, up to $n = 5$. We observe at iteration 5, with $N_{known} = 540$ graphs now, the classification is quite poor as the "Predicted" box plots are similarly distributed. The trend lines for the two medians also convey this point, with substantial decreases until the fifth iteration. This behavior is perhaps expected because we are not adding any new data from the initial $\mathcal{G}^1_{known}$, and the number of graphs for training decreases.

Assessing the outcome at the end of iteration 4 (since the fifth did not classify well), there are a total of 2,920 graphs in the final "good" set with a median of 3×10^{-4}, substantially lower than the initial classification median of 2×10^{-1}. Here we assume a designer would evaluate $J(G_i)$ for all the graphs in "Predicted 1"; thus, there would be a total of 11,029 graphs that were optimized. For this final set, *all* of the top 10, 87 of the top 100, and 727 of the top 1000 graphs still remain at 25% the computational cost of complete enumeration. Furthermore, compared to randomly sampling 11,029 graphs from $\mathcal{G}_{all}$, the IC model results greatly exceed the expected means of 2.25, 22.51, and 225.13 for the top 10, 100, and 1000 graphs being included in the random set, respectively.

To better understand the statistical significance of this iterative approach, five more randomized runs were completed with different samplings of $\mathcal{G}^1_{known}$. Over these runs, the average graphs that would be known or "optimized" was 11282.2 graphs (with a 492.8 standard deviation). For the top 100 graphs, 88.2 (2.5) remained compared to an expected value of 22.96. Finally, for the top 1000 graphs, 751.2 (39.5) remained compared to an expected value of 229.6. Furthermore, the median values of the "Known 1" and "Predicted 1" graph sets for each iteration are shown in Fig. 4.6 averaged over the six runs. Here we still see a decrease in the median values indicating the IC approach is narrowing down the set of potentially "good" graphs to the higher percentiles of

performance. In fact, at iteration 4, all medians for "Predicted 1" are better than the 90th percentile, with the mean closer to the 95th. However, as previously discussed in Fig. 4.5, the benefits from the 5th iteration are minimal.

As indicated in Fig. 4.4, there are expected trade-offs when attempting to reduce computational costs by decreasing the required evaluations of $J(G_i)$ and GDL tasks. Therefore, an additional study was conducted only reducing the starting $\mathcal{G}^1_{known}$. Initialized with only 10% of $\mathcal{G}_{all}$, the iterative approach now concluded with 8118.2 (969.1) graphs on average needing to be optimized per Eq. (4.1). Summarizing the results of five random runs, 9.2 (0.4) of the top 10 graphs remained, 85.2 (6.3) of the top 100 graphs remained, and 715.2 (76.9) of the 1000 graphs remained. Another difference between the different starting $\mathcal{G}_{known}$ sizes (i.e., 20% vs. 10%) is total iterative classification cost, including training the various GDL models at each iteration. The difference here is 45 vs. 36 minutes between 20% vs. 10% $\mathcal{G}^1_{known}$, respectively. Since both summarized top graph outcomes are comparatively similar, additional investigations into using fewer graphs for $\mathcal{G}^1_{known}$ are critical to understanding trade-offs related to the total computational costs of the design study to meet different designer goals.

4.3 Conclusion

The results showed that IC-GSBD can be used as an effective and efficient method for narrowing the graph sets for this case study. In the first trial, we established a baseline to determine model performance without any added features. The baseline model performed very well, with an accuracy of 79.1% while training on 72% of the data. After adding the additional node features, that accuracy increased to 85%, proving that feature engineering is a useful tool in helping separate the graphs from one another.

The following trial was then conducted to determine the optimal number of epochs and *known* set size. After testing various known set sizes, we were able to determine that the optimal number of epochs was 600. Using a similar approach, we were also able to determine an optimal N_{known} of 20%.

Using the information from those first trials, the iterative classification trial was then conducted, where the model identified 9.0 of the top 10, 88.2 of the top 100 graphs, and 751.2 out of the top 1000 at 25% the computational cost of complete enumeration. This shows that not only does IC work, but implementing AI/ML into the down-selection process significantly reduces computational cost. The original study in [17] took approximately 8 hours to complete, encompassing graph dataset creation through synthesis and analysis. In contrast, by leveraging the IC-GSBD framework, we were able to train on 5% of the data in just 36 minutes. It is important to note that this time, measurement excludes dataset creation, focusing solely on the training process.

This reduction in time highlights the efficiency of the IC-GSBD approach. By utilizing a fraction of the data for training, the framework can rapidly identify high-performing solutions without compromising accuracy. This efficiency accelerates the design process and enables the exploration of more complex design spaces within feasible timeframes, thus significantly enhancing productivity and innovation in engineering design tasks.

In the next chapter, we demonstrate this same approach in a different engineering field on a much different dataset to demonstrate the versatility of IC-GSBD.

Chapter 5

Case Study 2: Down-Selection of Aircraft Thermal Management Systems[4]

> "Efficiency is doing things right; effectiveness is doing the right things."
>
> *–Peter Drucker*

In this chapter, we extend the application of the Iterative Classification for Graph-Set-Based Design (IC-GSBD) framework to the down-selection of aircraft thermal management systems (TMS). Aircraft TMS are critical for maintaining optimal operating conditions and ensuring the reliability and performance of various aircraft components. Previous work in Ref. [25] showed that enumerative methods for a problem like this are computationally expensive due to the multiple objectives the systems have to meet; will the graph compile and simulate? These objectives are explained in further detail in Section 5.1.2, but determining these objectives is computationally expensive. This dataset also does not contain *optimal* solutions because not all of the solutions compiled successfully with their given parameters, and of those that did compile, only a large random sampling of parameters was evaluated rather than a systematic optimization [25]. Lastly, this dataset contains directed graphs, whereas in the previous case study, they were undirected. Therefore, this case study is another example to demonstrate the capabilities of IC-GSBD.

This chapter explores how IC-GSBD can be utilized to effectively narrow down this large set of potential TMS designs to a subset of useful solutions. We begin by providing an overview of the dataset and the methods used to generate the graph representations of TMS from Ref. [25]. We then discuss the performance metrics and optimization objectives used to evaluate these systems and initially determine what are promising options. Next, we detail the trials conducted to estab-

[4]Parts of this chapter are based on Ref. [52]

lish a baseline, perform feature engineering, and determine the optimal parameters for iterative classification.

The practical application of IC-GSBD in this case study highlights its potential to reduce computational costs and enhance decision-making efficiency in the design of aircraft TMSs.

5.1 Background

Graph-based models can represent TMS architectures using directed labeled graphs, depicting physical/functional components as vertices and connections as edges Ref. [25], as illustrated in Figure 2.6. Here, we will briefly review the methods from Ref. [25, 117] that generate and evaluate these graphs that are the focus of the case study.

5.1.1 Enumerative Methods with Network Structure Constraints

Enumerative techniques for graph theory involve creating a comprehensive catalog of *all* graphs that meet specific criteria. This complete catalog is often referred to as a graph structure space. The original study in Ref. [25] used multiple enumeration methods to ensure that all directed labeled graphs of interest were generated. With efficient enumeration graph algorithms, the relevant graph structure spaces can be determined (e.g., including hundreds of thousands of unique graphs).

To help improve the usefulness of the generated graphs, *Network Structure Constraints* (NSCs) were included in the generation and post-filtering steps. These include the direct connection NSC, which restricts connections between certain vertex types; the line-connectivity NSC, specifying permissible connections between different vertex types; and the multi-edges NSC, which ensures each edge in the graph is unique. Additionally, the intermediate component NSC identifies specific vertex types within a path, the downstream vertex NSC determines vertex relationships in a directed path graph, and the vertex-subgraph NSC manages vertex distribution across different subgraphs. Each constraint contributes significantly to the topologically valid and usefulness of the generated graphs [27, 28]. However, even with these NSCs, there are still valid graphs that are not valid

models of the physically realistic TMS architectures, which can be determined through physics-based analysis.

5.1.2 Analysis of Physics-Based Models

The graphs from the previous section now serve as data objects for constructing detailed simulation models in Modelica, enabling engineers to assess the performance and realizability of the TMS architecture. This process entails mapping each vertex in the graph to a specific Modelica component model, transforming the abstract graph representation into a text-based Modelica model file that can be compiled and simulated to help understand how the TMS will perform under different conditions.

Each TMS component (e.g., turbines, compressors, heat exchangers, etc.) is linked to a physics-based model capturing its behavior based on fundamental physical principles and data. For instance, a basic air cycle machine (ACM), as shown in Figure 2.6a, consists of two heat exchangers (HEX), a compressor (ACP), a turbine (TUR), and a fan (FAN). This system incorporates two primary air sources: ram air (RSO) and bleed air (BSO). The RSO utilizes external air that is generated by the aircraft's forward motion and is considerably cooler than the air inside the aircraft at high altitudes to cool various components and engine operations. Bleed air, on the other hand, is sourced from the engine's compressor section and plays a vital role in the aircraft's environmental control and anti-ice systems. Other components are also utilized but are not a part of every solution, such as a water separator (WSP), and a pressure regulating valve (PRV). Most common TMSs work by taking incoming ram air and passing it through the heat exchangers to help cool the engine bleed air as it passes from the compressor to the turbine back to the bleed air sink. Now observing Figure 2.6b, we can observe how the basic ACM can be represented as a graph, where the edges show the various air flows.

5.1.3 Dataset Overview

The dataset used in this study is from Ref. [25], where an aircraft TMS is tasked with controlling the temperatures of two hypothetical loads: a flight control heat load (HLF) and a radar heat

load (HLR). The designated target temperatures are 343 K for the flight control and 300 K for the radar. A multi-objective utility function is employed to rank and help select the most suitable architectural designs. This function considers various goals, including keeping thermal loads within specified temperature ranges. The utility function is defined as follows:

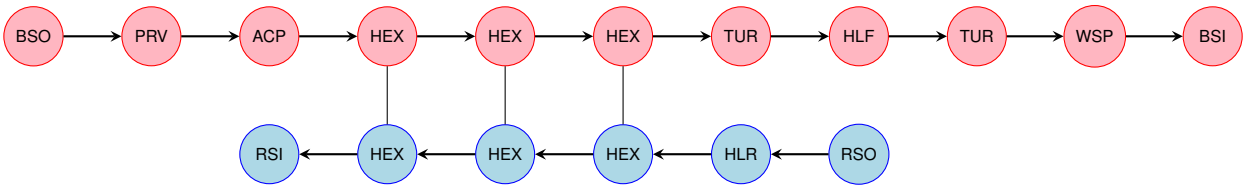
$$\text{minimize: } J = w_1 J_1 + w_2 J_2 + w_3 J_3 + w_4 J_4 + w_5 J_5 \tag{5.1}$$

$$= w_1(\bar{T}_{\text{HLF}} - 343K) + w_2(\bar{T}_{\text{HLR}} - 300K) + w_3 C_{\text{total}}$$

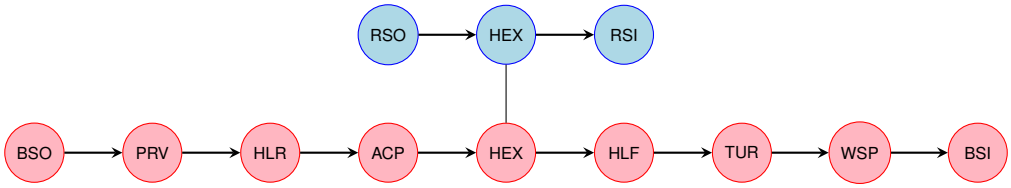
$$+ w_4 \dot{m}_{\text{BSI}} + w_5 \dot{m}_{\text{RSI}} \tag{5.2}$$

where $\bar{T}_{\text{HLF}}$ represents the average temperature of the flight control heat load, $\bar{T}_{\text{HLR}}$ is for the radar heat load, C_{total} denotes the total heat capacity of the heat exchangers, and $\dot{m}_{BSI}$ and $\dot{m}_{\text{RSI}}$ are the mass flow rates of the bleed air and ram air, respectively. Nominal weights have been established to provide a singular value for ranking different architectural solutions. Figure 5.1 displays six more TMS examples with varying numbers of heat exchangers (HEX) and configurations. In both Figure 5.1a and Figure 5.1b were simulatable solutions with (a) having $J = 469.9$ and (b) having $J = 719.04$. Figure 5.1c is a graph that did not compile at all and, therefore, has no value for J. Figure 5.1d, is an example of a graph that did compile but did not simulate and also does not have a value for J. Lastly, Figure 5.1e and Figure 5.1f are the graphs that did simulate and are the lowest utility values, $J = -167.82$ and $J = 5.33 \times 10^3$ respectively.

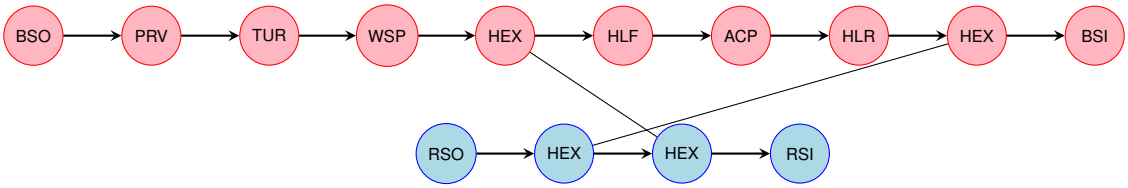
Employing the graph enumeration techniques outlined in Section 2.2.2 to generate a set of useful, non-isomorphic solutions based on the NSCs and in the previous section, a total of 32,612 potential architectures, represented as graphs, were evaluated. Of the considered graphs, 5,585 were successfully compiled, and among these, 2,098 could be simulated. Simulatability and best performance $J(G_i)$ were determined by trying to simulate compiled models using 200 parameter sets. If at least one parameter set produced a valid result, then it is considered simulatable, and the lowest J value from the valid results is assigned to $J(G_i)$. In Figure 5.2, the 2,098 simulatable



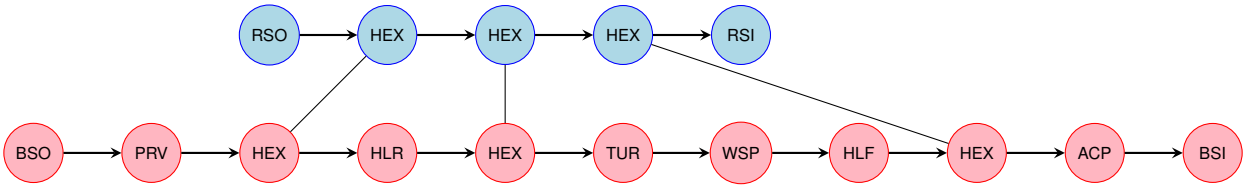
(a) A *simulatable* 17-node directed graph representing a TMS with three heat exchangers and $J = 469.9K$



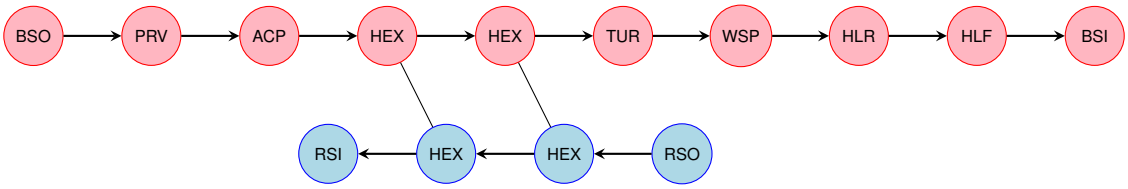
(b) A *simulatable* 12-node directed graph representing a TMS with a single heat exchanger and $J = 719.04K$



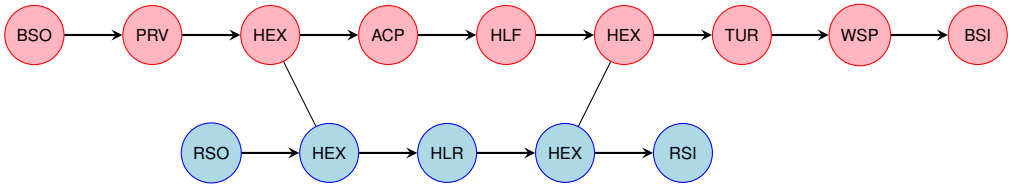
(c) A 14-node directed graph representing a TMS that *did not compile*



(d) A 16-node graph representing a TMS that *compiled*, but did not *simulate*



(e) A 14-node directed graph representing a TMS that had the lowest value of $J = -167.82K$



(f) A 14-node directed graph representing a TMS that had the highest value of $J = 5.33 \times 10^3 K$

Figure 5.1: Two more examples of TMSs with varying numbers of heat exchangers but also exhibiting other TMS components

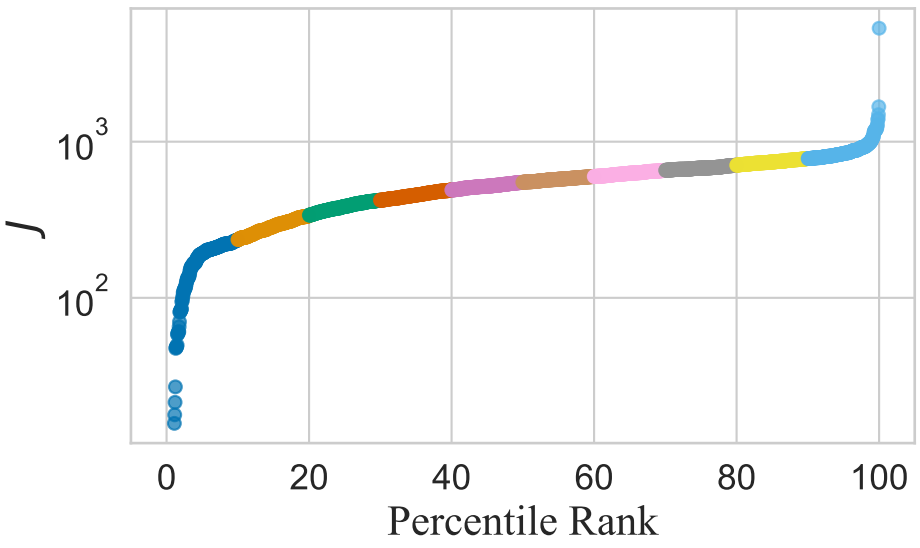


Figure 5.2: The utility values J for the 2,098 simulatable solutions where each color represents a percentile group

solutions are plotted to show the distribution of values across the dataset, where the highest value for J is 5,328.7, and the lowest is -167.8.

In the next section, we discuss the trials conducted on the dataset, beginning with multi-label classification to determine *compilability* and *simulatability*. Then, we use the information from the first trial and employ it in the second, where we explore the iterative classification of the dataset.

5.2 Trials

In the subsequent section, the tests conducted on the dataset outlined in the previous section will be examined alongside their corresponding outcomes. Commencing with the establishment of a baseline for multi-label classification, emphasis will be placed on ascertaining the ideal number of epochs and the magnitude of N_{known}. Our primary aim is to leverage the model to derive these values and furnish performance metrics, thereby determining the suitability of feature engineering as a viable direction. We then apply the values from these tests to the iterative classification test.

Table 5.1: The model metrics for different N_{known} averaged over the seven independent runs.

Known Size %	Mean Accuracy	Mean Precision	Mean Recall	Mean F1	Mean AUC
20	0.915	0.947	0.854	0.864	0.860
10	0.909	0.954	0.838	0.853	0.845
5	0.902	0.940	0.813	0.840	0.845
2.5	0.881	0.930	0.760	0.790	0.825
1.25	0.866	0.915	0.703	0.744	0.824

1: Multi-Label Classification to Determine Compilation and Simulatability

We first focus on developing a Geometric Deep Learning (GDL) model for the first scenario from Section 3.1.2 as understanding if a graph will compile and simulate is the first critical step in their evaluation.

Understanding Model Performance Through Known Set Size and Training Epochs

The initial step focuses on identifying the desired size of the $\mathcal{G}_{known}$ and the appropriate number of epochs for training our model. Given the need to balance model efficacy and data collection costs, establishing these parameters at the outset is essential. This first experiment explores models using five different N_{known}, each with seven independent runs for statistically significant results. The optimal dataset size and number of training epochs can be established by gathering and analyzing the data from these iterations and training runs. This determination will be based on charting the average training loss observed in each iteration. Additionally, the decision-making process will consider other vital metrics, such as the mean F1 score and the aggregate accuracy when the model is applied to the unseen data.

The metrics for each iteration and run of the model are detailed in Table 5.1. It is evident from these results that we can produce a performant model even when reduced to smaller values of around 5% of N_{all}. Upon reviewing Table 5.2 and Table 5.3, we find a detailed analysis of precision, recall, and F1 scores for comparability and simulatability, respectively. It becomes apparent that label l_1 maintains robust performance throughout all the explored N_{known}. This

Table 5.2: Precision, Recall, and F1 scores for the *compile* labels.

	Will not compile (l_1)			Will compile (l_2)		
Known Size %	Precision	Recall	F1	Precision	Recall	F1
20	0.948	0.965	0.957	0.818	0.743	0.778
10	0.941	0.968	0.954	0.819	0.708	0.759
5	0.937	0.964	0.951	0.800	0.688	0.740
2.5	0.941	0.948	0.945	0.740	0.714	0.727
1.25	0.883	0.982	0.930	0.810	0.373	0.510

Table 5.3: Precision, Recall, and F1 scores for the *simulate* labels.

	Will not simulate (l_3)			Will simulate (l_4)		
Known Size %	Precision	Recall	F1	Precision	Recall	F1
20	0.981	0.984	0.982	0.758	0.722	0.739
10	0.979	0.983	0.981	0.744	0.692	0.717
5	0.974	0.988	0.981	0.784	0.610	0.686
2.5	0.959	0.992	0.975	0.769	0.385	0.512
1.25	0.964	0.986	0.975	0.702	0.471	0.562

strong performance may be due to a large portion of the data being labeled 1 within this category. Additionally, when there's a considerable imbalance in the data for specific labels, such as l_2 and l_4, the model can exhibit issues with very small N_{known}. Finally, by examining the Matthews Correlation Coefficient (MCC) for each label pairing as presented in Table 5.4, the model exhibits weaker performance with smaller known dataset sizes, consistent with expectations.

Now, by examining Figure 5.3, which illustrates the average training loss for each iteration across different runs, we observe that the loss remains relatively consistent between 20% and 2.5% dataset sizes. Furthermore, the training loss seems to slow down around 300 epochs in most cases.

Integrating these insights enables an informed selection of the N_{known} and epoch count for subsequent experiments. Based on this data, it was decided to utilize 5% of the dataset and conduct training over 300 epochs. This decision is supported by the data in Table 5.1, which indicates

Table 5.4: Matthews Correlation Coefficient for each N_{known} and label pairing

Known Size %	l_1 & l_2	l_3 & l_4
20	0.736	0.722
10	0.717	0.699
5	0.693	0.673
2.5	0.671	0.524
1.25	0.496	0.552

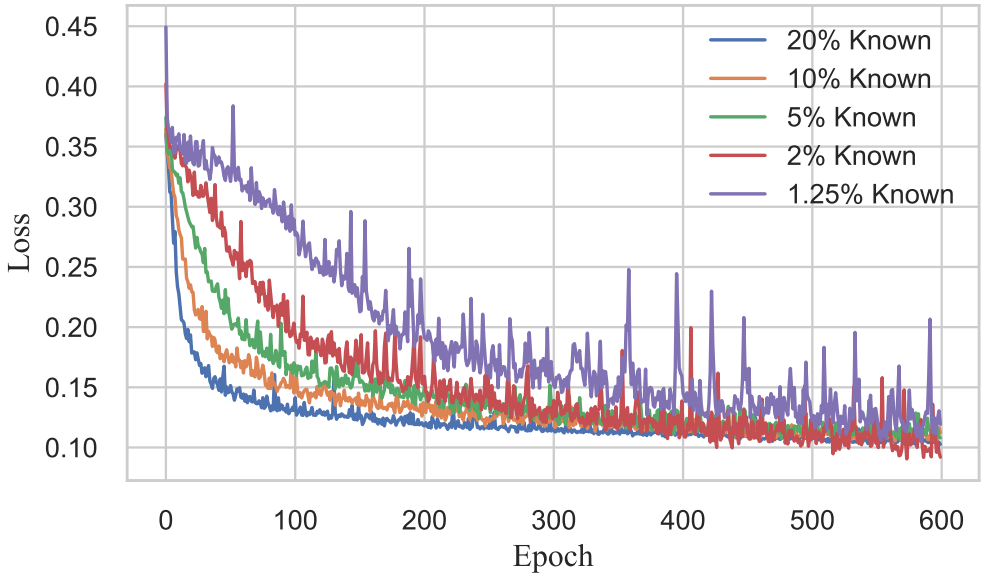


Figure 5.3: Average training loss for each iteration per run

that the average accuracy through to the AUC remains high for this dataset size. Furthermore, as detailed in both Table 5.2 and Table 5.3, the 5% dataset size demonstrates strong performance even when analyzed by an individual label. Lastly, the 5% known set size MCC value shows relatively high prediction strength for all labels. The choice of 300 epochs is justified by observing that the loss for each model tends to level off around this point, with the differences between them being negligible.

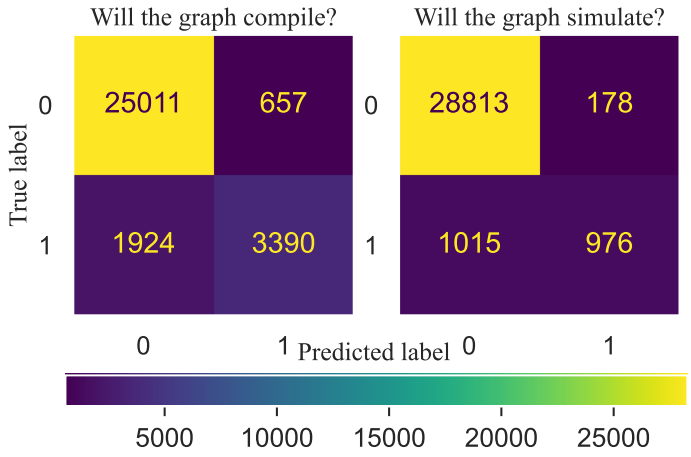
Improving Model Performance with Feature Engineering

As addressed in Section 3.2, here we intend to demonstrate the effects of feature engineering with the TMS dataset utilizing PCA. Using the values from the confusion matrix shown in Figure 5.4a, the model achieved an accuracy of 91.67% and 96.15%. The MCC values are 68.5% and 62.7%, respectively. The confusion matrix for the model with the additional feature, detailed in Figure 5.4b, reveals significant improvements: the model's accuracy reached 98.38% and 97.84%, and the MCC 94.2% and 80.6%. These enhancements indicate that the model achieved superior predictive performance on the same $\mathcal{G}_{unknown}$ dataset, with all metrics showing better results. Consequently, we will incorporate this feature in future model constructions.

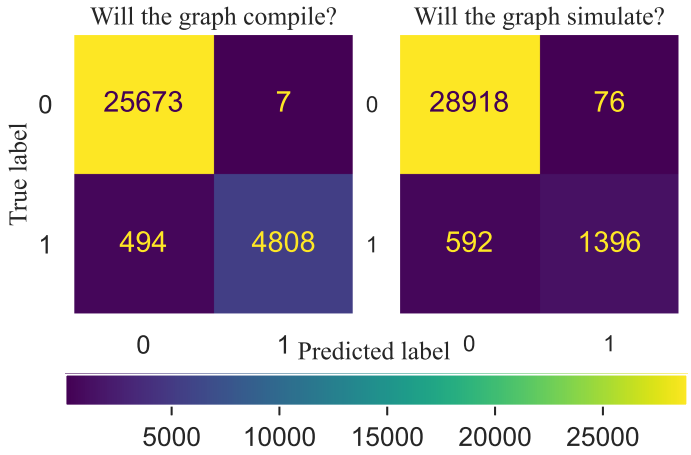
Iterative Down-selection Based on Graph Performance

In addition to compiling and simulating the TMS graphs, we are also concerned with which ones perform the best concerning Equation (5.2). In this section, we use the iterative classification approach from Section 3.6.2 to down-select a subset of graphs from $\mathcal{G}_{all}$ with higher median performance (i.e., potentially good graphs).

Here, we start with 40% of the graphs already evaluated as presented in Figure 5.6. We see the ranking of values from each set from each iteration compared to the rankings from the original dataset $\mathcal{G}_{all}$. Initially, the far-left bar plot illustrates the distribution of $J(\mathcal{G}_{all})$, ranking its values from 1 (the best J value) to N_{all}. The subsequent two bar plots demonstrate the division of $\mathcal{G}_{known}$ according to the median performance classification threshold in the first iteration. This iteration's last two bar plots represent the GDL model's predictions of 1s and 0s. It's important to note that some incorrectly classified graphs exist, yet the medians remain distinct and aligned with their respective "Known" categories. At the end of iteration 4, the final "good" set had 383 graphs, with a median of 297.43, much lower than the initial classification of 549.53. For this final set produced in iteration 4, the model predicted 9 of the top 10, 83 of the top 100, and 163 of the top 200, with an overall count of 832 out of 2098 evaluated graphs.



(a) Baseline confusion matrix.

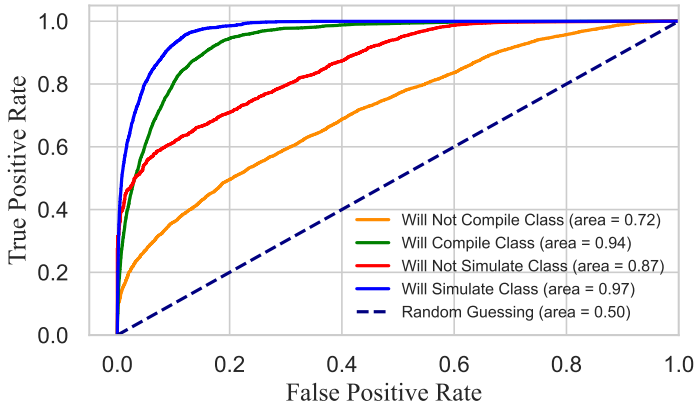


(b) With the addition of the PCA features.

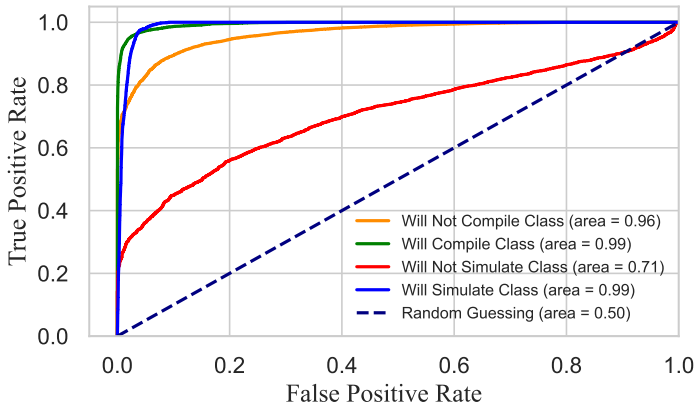
Figure 5.4: Confusion matrices on $\mathcal{G}_{unknown}$ for models with different feature sets

An additional nine runs were conducted to gain a deeper insight into the statistical significance of this iterative method, each with a different random subset from $\mathcal{G}_{known}$. Across these runs, the average number of optimized graphs was 875.7, with a standard deviation of 34.3. Specifically, the average for the top 10 graphs was 4.75 with a standard deviation of 1.48; for the top 100, it was 69 with a standard deviation of 4.95. Consequently, these results show good performance when identifying these desired sets.

Finally, in Figure 5.7a, we can see both the "Known 1" and "Predicted 1" categories averaged over the ten runs out to four iterations. In the third and fourth iterations, the model achieved poor classification results exhibited by the significant variation in median utility values, primarily due



(a) Baseline ROC curve.



(b) With the addition of the PCA Features.

Figure 5.5: ROC curves on $\mathcal{G}_{unknown}$ for models with different feature sets.

to the very small training set size at this point. But due to the supplementation of predicted ones into the $\mathcal{G}_{known}$ during recurrent iterations, the "known" set was in fact narrowed down to the top 5th percentile. Figure 5.7b displays the cumulative distribution of values across each iteration. As evident in conjunction with Figure 5.7a, it can be observed that with every iteration, the values in both the "Known 1" and "Predicted 1" sets diminish, increasingly concentrating on the better-performing graphs. In this case, the observed reduction in median values suggests that the *iterative GDL* approach is effectively refining the selection of potentially "good" graphs, focusing on those that rank in the higher percentiles of J.

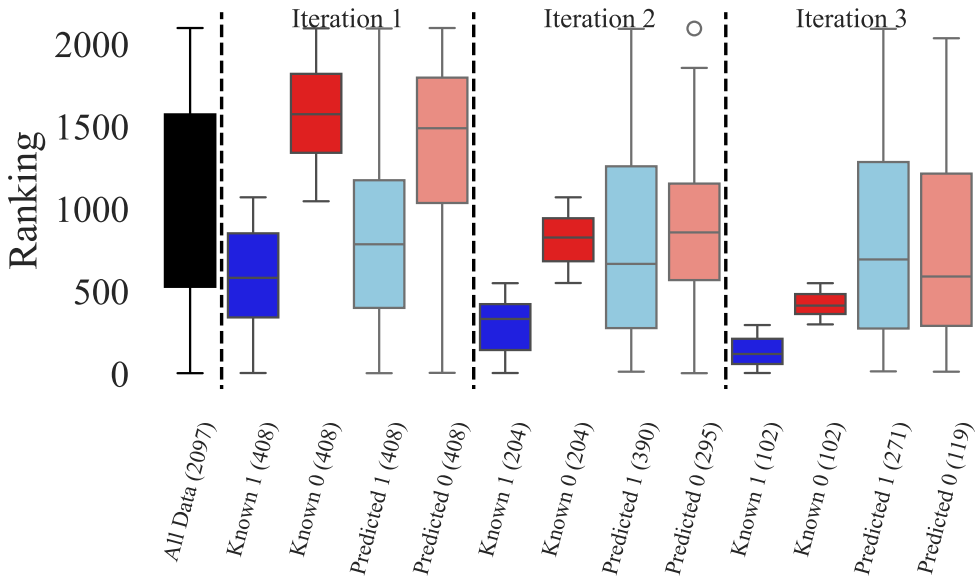
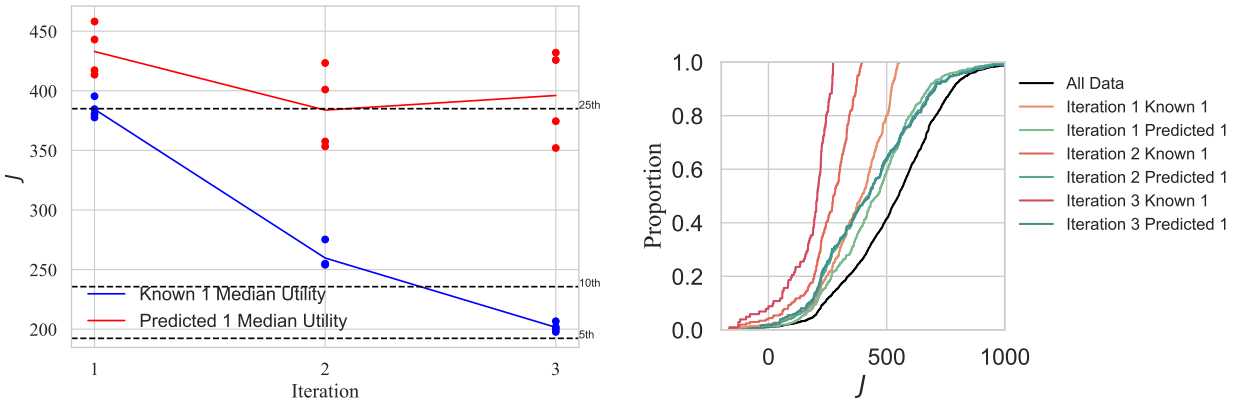


Figure 5.6: Three cycles of the method outlined in Section 3.6.2 are applied to retrain a GDL model, each cycle using progressively refined subsets of graphs to minimize performance issues. These subsets are indicated by their respective graph counts. Additionally, all data point rankings are distributed for comparative analysis.

5.3 Conclusion

This case study utilized the Iterative Classification of Graph-Set-Based Design framework (IC-GSBD) to classify and refine graph-based aircraft thermal management systems (TMSs), aiming to identify feasible and superior solutions in an engineering design context. In the case study, all graph objects were known, but certain outcomes, such as model simulatability and performance, were computationally expensive to determine. Thus, the effectiveness of IC-GSBD in distinguishing "good" and "bad" graphs with minimal known outcome data was demonstrated. Moreover, incorporating the graph-based features of harmonic, betweenness, and eigenvector centrality, as well as spectral radius utilizing the principle component analysis (PCA) method, has enhanced several critical metrics.

Upon commencing our study by establishing a baseline for multi-label classification, we demonstrated that IC-GSBD can address problems with a single objective created by a weighted sum that contains multiple constraints. Subsequently, by incorporating additional node-level features through Principle Component Analysis (PCA), we enhanced the model's capability to distinguish



(a) Median utility values of the "Known 1" and "Predicted 1" sets averaged over ten runs.

(b) The cumulative distribution for a single run displays the proportion of values contained in each iteration.

Figure 5.7: Results from the iterative classification for down-selection study.

between "good" and "bad" solutions. Incorporating PCA into an approach such as this shows that there are ways to handle problems that have missing data, and since our goal is to reduce the computational cost, the low cost of utilizing PCA is minimal. After adding the PCA features to the nodes and determining a graph's compilability and simulatability, models have achieved over 97% accuracy while utilizing only 5% of the entire dataset ($\mathcal{G}_{all}$).

For the iterative down-selection based on graph performance trial, we observed that using around 40% of the data identified, on average, 7.3 (1.79) of the top 10 graphs and 75.5 (7.95) of the top 100. With the extreme computational costs associated with determining any of these outcomes on the set of graphs, the results demonstrate a potentially drastic reduction in total costs.

The outcomes presented serve as a step towards exploring general strategies for creating efficient GDL models for graph-based engineering design problems tailored for the classification and down-selection goals. Critical aspects still require further study, especially when it comes to smaller datasets and intelligently selecting which graphs to evaluate outcomes (rather than the pure random approach used in this paper).

Once again, there was a significant computational cost reduction, as the original study in Ref. [25] took several days. Not including the time it took for the graph dataset and physics model simulations, the down-select process took approximately 33 minutes, which is relatively insignificant overall.

Chapter 6

Conclusion and Future Directions

As systems engineering, system architecture, and engineering design continue to evolve, the integration of advanced computational techniques such as Graph-Set-Based Design (GSBD) holds significant promise for addressing complex, graph-centric design problems.

This dissertation introduced the methodological framework, Iterative Classification for Graph-Set-Based Design (IC-GSBD). We demonstrated that the integration of Geometric Deep Learning (GDL) and a Set-Based Design (SBD) approach could effectively address complex engineering design problems. The findings in this work provide key contributions to engineering design, specifically problems that involve designs with graph-based representations and that come with high computational costs if they were to be evaluated as possible solutions by humans or physics-based models. One such contribution is a framework that enhances the design process by leveraging the strengths of GDL for graph-based data and facilitates the efficient down-selection of potential design solutions, significantly reducing computational cost but also comes with the price of potentially losing promising solutions. Once this down-selection occurs, more detailed models and experts can further evaluate this new subset of useful solutions.

Through the two case studies, we demonstrated the effectiveness of Iterative Classification in narrowing down the top-performing solutions. The first case study focused on the down-selection of analog electrical circuits. We initiated our study with 43,249 undirected graphs, each representing a unique electrical circuit containing a maximum of six impedance subcircuits. Assigning a performance value $J(G_i)$ to each graph quantifies the disparity between desired and actual frequency responses where graphs with the lowest value of J were determined to be the better solutions. Calculating this performance value for each circuit required solving an expensive optimization problem with a set of constraints. Subsequently, we utilized a fraction of this dataset to train a GDL model. Impressively, the model efficiently identified 9 out of the top 10 viable solu-

tions within the dataset, accomplishing this task within a significantly reduced timeframe. Doing so gave us our initial insights into the potential capabilities of IC-GSBD.

The second case study explored the down-selection of aircraft thermal management systems (TMS). We began with 32,612 directed graphs, each representing a potential solution. The performance value J was the result of a utility function that sought to minimize the value of J based on balancing various weights, temperature ranges, and mass flow rates. This evaluation was accomplished by attempting to compile and simulate each graph using 200 parameter sets. This entire process was extremely expensive to determine that only 2,098 graphs were capable of simulating and their performance. We then took that same dataset and demonstrated that by using only 5% of the data, graph classification alone could determine with 98% accuracy which graphs would compile and which would simulate. Then, utilizing the performance values for the 2,098 graphs that simulated, we demonstrated that IC-GSBD is capable of narrowing down on average 7.3 out of the top 10 solutions in a fraction of the time it took the previous study.

Both of these studies not only demonstrated the ability of IC-GSBD to achieve high classification accuracy along with efficient computational costs but also that if specific data is not available, the graphs themselves are capable of producing the needed data to fill the gaps in assessment based on performance, thus aiding in the classification process. By doing this, we demonstrated not only the capabilities of IC-GSBD but also another example of the immense potential that machine learning techniques have in engineering design.

The cost reductions shown in this work will not only give stakeholders a more informed and timely decision based on their requirements but also enable them to make more informed decisions by allowing them to evaluate more options, i.e., give them the ability to evaluate even larger datasets due to the time savings. This outcome is in support of the concept of Agile systems engineering by increasing efficiency and effectiveness through:

- **Combining SDB & GDL Techniques** — Without altering the SBD workflow, IC-GSBD introduced a machine learning framework, thus providing a potential level of automation to the iterative process. Using graph-based representations and Geometric Deep Learning,

the capability to handle complex engineering system datasets is enhanced, and efficiency is increased through reduced computational cost, thus accelerating the development process. Multi-label classification was also included to help aid in the decision-making process by allowing the evaluation of multiple criteria.

- **Prediction of Complex Graph-based Engineering Systems** — IC-GSBD demonstrated the capability of analyzing and predicting complex engineering systems represented as graphs (and only with the graph data) through the use of GDL in the two case studies.
- **Demonstrated Value of Additional Derived Graph-Based Features** — Model performance was shown to increase with additional graph-based features.

Returning to the topic from Section 1.2 regarding AI/ML in systems engineering and the human-in-the-loop. IC-GSBD not only demonstrated the positive impact of adding AI/ML to the design process but also showed the need for a human to manage it. In the second case study, specifically Section 5.2, we can see that the model did not perform as well as hoped after the first iteration. A designer would be required to decide what to do next, whether to collect more or alter any of the data, discard the other iterations, or rethink their entire approach. The point is more than just relying on the model to make all final decisions. This further supports the Systems Engineering Research Center's roadmap towards Human-Machine co-learning. These case studies have demonstrated that both a designer and AI/ML can work together in the down-select process to come up with a set of useful alternatives to present to stakeholders, knowing that not only the data supports the decision but the model as well.

We have extensively discussed the advantages of using IC-GSBD, but it is not without its drawbacks:

- As dataset size and complexity increase, achieving high accuracy in the down-select process may require more computational resources and time.
- If the user prioritizes computational efficiency, there may be a tradeoff with a reduction in machine learning model accuracy.

- A Set-Based Design approach encourages the exploration of multiple design alternatives simultaneously, which is beneficial for finding diverse solutions but can be very resource-intensive due to the large amounts of data involved.

- GSBD provides the flexibility to explore a wide range of design options, which may require time and resources, so focusing on the "best" alternative only can be more efficient, but it may lead to missing out on a better solution.

The other drawback of this framework, in its current state, is that it is limited to the early conceptual design phase. GSBD (as does SDB) requires the maintenance and evaluation of multiple design sets, which can become impractical and resource-intensive as designs progress to detailed development and production stages. Lastly, it requires competent knowledge of coding, machine learning, and data analysis in order to be implemented and interpreted correctly.

6.1 Future Work

The findings here provide a strong foundation for future research into applying AI/ML to graph-based system engineering processes that require expensive computations. The following areas represent key directions for future research and development:

1. **Enhanced Model Capabilities** — Future research could focus on improving the accuracy and efficiency of GDL models. This work includes exploring advanced neural network architectures, optimizing hyperparameters, and incorporating additional graph-based and node-based features. The exploration of other Python modules and incorporating them into a model that better optimizes the hyperparameters based on the dataset.

2. **Scalability & Computational Efficiency** — As design problems become more complex and datasets become larger, it is essential to develop scalable solutions that can handle larger datasets and more intricate graph structures. This goal means having a system, whether computer hardware or software-based, that can accommodate growth in data volume or other complexity measures. This outcome can be accomplished through distributed computing,

which divides a computational task across multiple machines or processors, or through parallel processing, which involves performing multiple computations simultaneously. It is our intent for future work to incorporate these tools to further demonstrate the computational efficiency when added to the IC-GSBD methodological framework.

3. **Interdisciplinary Applications** — Expanding the utilization of the IC-GSBD framework to other domains beyond analog circuits and thermal management systems is an area of potential exploration. The two case studies outlined here are preliminary demonstrations of the capabilities of IC-GSBD. Extending this framework to address Model-Based Systems Engineering (MBSE) challenges with datasets based on the Systems Modeling Language (SysML), Unified Modeling Language (UML), or Unified Architecture Framework (UAF) offers promising prospects. MBSE typically encompasses complex and extensive datasets. Leveraging GDL and iterative classification could empower modelers to extract features utilizing techniques like node classification or link prediction, as well as facilitate iterative analysis of intricate data. Furthermore, GSBD could systematically explore and assess multiple design alternatives, ultimately facilitating the identification of optimal solutions.

4. **User-Friendly Tools and Interfaces** — The development of more user-friendly tools with graphical user interfaces (GUIs) aimed at facilitating the application of the IC-GSBD framework to graph-based problems. These tools should allow users to input their graph-based dataset, specify the number of node features, or opt to utilize a PCA method for creating node features. The program could then analyze the graphs and provide insights into the dataset using large language models (LLM) by providing feedback to the user. Notably, efforts to aid users in the graph enumeration process are already underway [117], and integrating the IC-GSBD methodological framework with such tools would enable comprehensive end-to-end utilization.

5. **Multi-Objective Problems** — Case Study 2 discussed multiple objectives for the aircraft TMS dataset, although only a single weighted objective was used for IC. Many engineering

design problems have an extensive array of objectives. For example, in the circuit problem from Case Study 1, we have the option to include objectives for the number of impedance elements, the overall circuit complexity, or cost values for the various components. Our future goal is to ascertain not only the effectiveness of IC-GSBD as a tool for the down-select process but also to gauge its capacity for handling multiple, potentially competing, measures of graph value.

6. **Determine a Trained Models' Complexity Limit** — In Case Study 1, the model was trained on graphs ranging from 6 nodes to 20 nodes. A potential future effort could be to determine if the same trained model can be used to predict graphs larger than 20 nodes that would have otherwise been evaluated by the same techniques described in Section 4.1.1 and Section 4.1.2. This study would answer the question of whether we need to consistently create new GDL models for new data with larger graphs and to what extent the new data is needed for any retraining or transfer learning. Such an approach would enable users to gauge the model's capability based on the data it has already processed.

7. **Improve Initial $\mathcal{G}_{known}$ Sampling** — In the two case studies in this work, the initial $\mathcal{G}_{known}$ set was created by randomly sampling from $\mathcal{G}_{all}$. A potential future effort can be to improve this process by utilizing the information within the dataset, such as increasing the variety of sampled (known) graphs so that the model training can see a better variety of possible solutions. This approach can also include increasing the diversity in the type and number of components each graph has in the initial set. This approach can increase the trained model's ability to make an accurate prediction, thus resulting in a better decision-making tool for a designer.

Bibliography

[1] G. Pahl, W. Beitz, J. Feldhusen, and K.-H. Grote, *Engineering Design: A Systematic Approach*, 3rd ed. London: Springer London, 2007, doi: 10.1007/978-1-84628-319-2

[2] J. D. Summers and J. J. Shah, "Mechanical engineering design complexity metrics: Size, coupling, and solvability," *Journal of Mechanical Design (1990)*, vol. 132, no. 2, 2010, doi: 10.1115/1.4000759

[3] W. ElMaraghy, H. ElMaraghy, T. Tomiyama, and L. Monostori, "Complexity in engineering design and manufacturing," *College International pour la Recherche en Productique Annals*, vol. 61, no. 2, pp. 793–814, 2012, doi: 10.1016/j.cirp.2012.05.001

[4] W. F. Lawless, R. Mittu, D. A. Sofge, T. M. Shortell, and T. A. McDermott, in *Systems engineering and artificial intelligence*, ser. Springer eBook Collection. Cham, Switzerland: Springer, 2021, doi: 10.1007/978-3-030-77283-3

[5] J. Stokes, A. Sullivan, and N. Greene, "U.S.-china competition and military AI: How Washington can manage strategic risks amid rivalry with Beijing," July 25, 2023, accessed: 2024 May 29. url: https://www.cnas.org/publications/reports/u-s-china-competition-and-military-ai.

[6] E. B. Jania, ""AI weapons" in China's military innovation," April 2020, accessed: 2024 May 29. url: https://www.brookings.edu/articles/ai-weapons-in-chinas-military-innovation/.

[7] A. Press, "Biden: China should expect "extreme" competition from US," February 7, 2021, accessed: 2024 May 29. url: https://apnews.com/biden-china-should-expect-extreme-competition-from-us-8f5158c12eed14e002bb1c094f3a048a.

[8] M. Dwyer, "AI principles and the challenge of implementation," November 19, 2019, access: 2024 May 29. url: https://www.csis.org/analysis/ai-principles-and-challenge-implementation.

[9] "Research roadmaps 2019-2020," Systems Engineering Research Center, Tech. Rep., 2019.

[10] Z. Bi and X. Wang, *Computer aided design and manufacturing*, 1st ed., ser. Wiley-ASME press series. Hoboken, NJ: John Wiley & Sons, Inc., 2020, doi: 10.1002/9781119667889

[11] L. Caudill and A. Barnhorn, "60 years of CAD infographic: the history of CAD since 1957," 2018, url: https://partsolutions.com/60-years-of-cad-infographic-the-history-of-cad-since-1957.

[12] J. Sobieszczanski-Sobieski, A. J. Morris, and M. J. L. v. Tooren, *Multidisciplinary design optimization supported by knowledge based engineering*. Chichester, UK: John Wiley & Sons, Inc., 2015, doi: 10.1002/9781118897072

[13] M. Sandberg, "Knowledge based engineering - in product development," Lelea University of Technology, Tech. Rep., 2003.

[14] D. K. Sobek II, A. C. Ward, and J. K. Liker, "Toyota's principles of set-based concurrent engineering," *Sloan Management Review*, 1999, accessed: 2024 May 29. url: https://sloanreview.mit.edu/article/toyotas-principles-of-setbased-concurrent-engineering/.

[15] B. Toche, R. Pellerin, and C. Fortin, "Set-based design: a review and new directions," *Design Science*, vol. 6, p. e18, 2020, doi: 10.1017/dsj.2020.16

[16] J. M. Borky and T. H. Bradley, *Effective Model-Based Systems Engineering*. Cham, Switzerland: Springer, 2019, doi: 10.1007/978-3-030-14134-7

[17] D. R. Herber, "Advances in combined architecture, plant, and control design," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2017, url: http://hdl.handle.net/2142/99394.

[18] D. Selva, B. Cameron, and E. Crawley, "Patterns in system architecture decisions," *Systems Engineering*, vol. 19, no. 6, pp. 477–497, 2016, doi: 10.1002/sys.21370

[19] R. M. Foster, "Geometrical circuits of electrical networks," *Transactions of the American Institute of Electrical Engineers*, vol. 51, no. 2, pp. 309–317, 1932, doi: 10.1109/T-AIEE.1932.5056068

[20] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 417–426, doi: 10.1145/3308558.3313488

[21] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," 2019, arXiv:1909.03496

[22] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 3, 2021, doi: 10.1145/3436877

[23] W. Yang, H. Ding, and D. Zhang, "New graph representation for planetary gear trains," *ASME Journal of Mechanical Design*, vol. 140, no. 1, p. 012303, 2018, doi: 10.1115/1.4038303

[24] C.-H. Hsu and K.-T. Lam, "A new graph representation for the automatic kinematic analysis of planetary spur-gear trains," *ASME Journal of Mechanical Design*, vol. 114, no. 1, pp. 196–200, 1992, doi: 10.1115/1.2916916

[25] D. R. Herber, J. T. Allison, R. Buettner, P. Abolmoali, and S. S. Patnaik, "Architecture generation and performance evaluation of aircraft thermal management systems through graph-based techniques," in *AIAA 2020 Science and Technology Forum and Exposition*, no. AIAA 2020-0159, Orlando, FL, USA, Jan. 2020, doi: 10.2514/6.2020-0159

[26] D. C. Arney and A. W. Wilhite, "Modeling space system architectures with graph theory," *Journal of Spacecraft and Rockets*, vol. 51, no. 5, pp. 1413–1429, 2014, doi: 10.2514/1.A32578

[27] D. R. Herber, T. Guo, and J. T. Allison, "Enumeration of architectures with perfect matchings," *ASME Journal of Mechanical Design*, vol. 139, no. 5, p. 051403, 2017, doi: 10.1115/1.4036132

[28] D. R. Herber, "Enhancements to the perfect matching approach for graph enumeration-based engineering challenges," in *ASME 2020 International Design Engineering Technical Conferences*, no. DETC2020-22774, Aug. 2020, doi: 10.1115/DETC2020-22774

[29] P. A. Macmahon, "The combinations of resistances," *Discrete Applied Mathematics*, vol. 54, no. 2, pp. 225–228, 1994, doi: 10.1016/0166-218X(94)90024-8

[30] M. W. Maier and E. Rechtin, *The Art of Systems Architecting*, 3rd ed. Boca Raton, FL: CRC Press, 2009.

[31] J. Taft, "A mathematical representation of system architectures," Battelle for the US Department of Energy, Pacific Northwest National Laboratory, Tech. Rep. PNNL-27387, Mar. 2018.

[32] M. Potts, P. Sartor, A. Johnson, and S. Bullock, "Hidden structures: using graph theory to explore complex system of systems architectures," in *Complex Systems Design and Management*, 2017.

[33] L. C. Schmidt, H. Shetty, and S. C. Chase, "A graph grammar approach for structure synthesis of mechanisms," *ASME Journal Mechanical Design*, vol. 122, no. 4, pp. 371–376, 1999, doi: 10.1115/1.1315299

[34] D. F. Wyatt, D. C. Wynn, J. P. Jarrett, and P. J. Clarkson, "Supporting product architecture design using computational design synthesis with network structure constraints," *Research in Engineering Design*, vol. 23, pp. 17–52, 2012, doi: 10.1007/s00163-011-0112-y

[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015, doi: 10.1038/nature14539

[36] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017, doi: 10.1109/MSP.2017.2693418.

[37] K. Atz, F. Grisoni, and G. Schneider, "Geometric deep learning on molecular representations," 2021, arXiv:2107.12375

[38] P. Gainza, F. Sverrisson, F. Monti, E. Rodola, D. Boscaini, M. M. Bronstein, and B. E. Correia, "Deciphering interaction fingerprints from protein molecular surfaces using geometric deep learning," *Nature*, vol. 17, pp. 184–192, 2020, doi: 10.1038/s41592-019-0666-6

[39] M. H. S. Segler, T. Kogej, C. Tyrchan, and M. P. Waller, "Generating focused molecule libraries for drug discovery with recurrent neural networks," *ACS Central Science*, vol. 4, no. 1, pp. 120–131, 2018, doi: 10.1021/acscentsci.7b00512

[40] S. Fedorova, A. Tono, M. S. Nigam, J. Zhang, A. Ahmadnia, C. Bolognesi, and D. Michels, "Synthetic data generation pipeline for geometric deep learning in architecture," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLIII-B2-2021, pp. 337–344, 2021, doi: 10.5194/isprs-archives-XLIII-B2-2021-337-2021

[41] A. H. Thiery, F. Braeu, T. A. Tun, T. Aung, and M. J. A. Girard, "Medical application of geometric deep learning for the diagnosis of glaucoma," 2022, arXiv:2204.07004

[42] I. Sarasua, J. Lee, and C. Wachinger, "Geometric deep learning on anatomical meshes for the prediction of alzheimer's disease," 2021, arXiv:2104.10047

[43] J. C. Wong, C. C. Ooi, J. Chattoraj, L. Lestandi, G. Dong, U. Kizhakkinan, D. W. Rosen, M. H. Jhon, and M. H. Dao, "Graph neural network based surrogate model of physics sim-

ulations for geometry design," in *IEEE Symposium Series on Computational Intelligence*, 2022, pp. 1469–1475, doi: 10.1109/SSCI51031.2022.10022022

[44] V. Krokos, S. P. A. Bordas, and P. Kerfriden, "A graph-based probabilistic geometric deep learning framework with online physics-based corrections to predict the criticality of defects in porous materials," 2022, arXiv:2205.06562

[45] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia, "Learning mesh-based simulation with graph networks," 2021, arXiv:2010.03409

[46] J. Park and J. Park, "Physics-induced graph neural network: an application to wind-farm power estimation," *Energy*, vol. 187, p. 115883, 2019, doi: 10.1016/j.energy.2019.115883

[47] G. Zhang, H. He, and D. Katabi, "Circuit-GNN: Graph neural networks for distributed circuit design," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 7364–7373, url: https://proceedings.mlr.press/v97/zhang19e.html.

[48] Y. Xiao, F. Ahmed, and Z. Sha, "Graph neural network-based design decision support for shared mobility systems," *ASME Journal of Mechanical Design*, vol. 145, no. 9, p. 091703, 2023, doi: 10.1115/1.4062666

[49] V. Ferrero, B. DuPont, K. Hassani, and D. Grandi, "Classifying component function in product assemblies with graph neural networks," *ASME Journal of Mechanical Design*, vol. 144, no. 2, p. 021406, 2021, doi: 10.1115/1.4052720

[50] T. Guo, D. R. Herber, and J. T. Allison, "Circuit synthesis using generative adversarial networks (GANs)," in *AIAA Scitech Forum*, Jan. 2019, doi: 10.2514/6.2019-2350

[51] A. Sirico and D. R. Herber, "On the use of geometric deep learning for the iterative classification and down-selection of analog electric circuits," *Journal of Mechanical Design*, vol. 146, no. 6, p. 051703, May 2024, doi: 10.1115/1.4063659

[52] A. Sirico Jr and D. R. Herber, "Geometric deep learning towards the iterative classification of graph-based aircraft thermal management systems," in *AIAA 2024 Science and Technology Forum and Exposition*, Orlando, FL, USA, Jan. 2024, doi: 10.2514/6.2024-0684

[53] H. Petroski, *To Engineer Is Human: The Role of Failure in Successful Design*. Vintage Books, 1992.

[54] D. J. Singer, N. Doerry, and M. E. Buckley, "What is set-based design?" *Naval Engineers Journal*, vol. 121, no. 4, pp. 31–43, 2009.

[55] G. S. Parnell, E. Specking, S. R. Goerger, M. Cilli, and E. Pohl, "Using set-based design to inform system requirements and evaluate design decisions," in *INCOSE International Symposium*, vol. 29, no. 1. Wiley Online Library, 2019, pp. 1123–1138, doi: 10.1002/j.2334-5837.2019.00609.x

[56] A. C. Ward, J. K. Liker, J. J. Cristiano, and D. K. S. II, "The second toyota paradox: How delaying decisions can make better cars faster," *Sloan Management Review*, 1995.

[57] M. N. Kennedy, *Product Development for the Lean Enterprise: Why Toyota's System is Four Times More Productive and How You Can Implement It*. Oaklea Press, 2009.

[58] R. Diestel, *Graph Theory*. Springer, 2017, doi: 10.1007/978-3-662-53622-3

[59] K. Munagala and A. Ranade, "I/o-complexity of graph algorithms," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '99. USA: Society for Industrial and Applied Mathematics, 1999, p. 687–694.

[60] J. A. Bondy and U. S. R. Murty, *Graph theory*, ser. Graduate texts in mathematics, 244. New York: Springer, 2008, doi: 10.1007/978-1-84628-970-5

[61] V. S. Borkar, V. Ejov, and G. T. Nguyen, *Hamiltonian Cycle Problem and Markov Chains*. Springer, 2012, doi: 10.1007/978-1-4614-3232-6

[62] D. R. Herber and J. T. Allison, "A problem class with combined architecture, plant, and control design applied to vehicle suspensions," *ASME Journal of Mechanocal Design*, vol. 141, no. 10, p. 101401, 2019, doi: 10.1115/1.4043312

[63] T. Guo, D. R. Herber, and J. T. Allison, "Reducing evaluation cost for circuit synthesis using active learning," in *ASME International Design Engineering Technical Conferences*, no. DETC2018-85654, 2018, doi: 10.1115/DETC2018-85654

[64] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, p. 84–90, May 2017, doi: 10.1145/3065386

[65] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng, "End-to-end text recognition with convolutional neural networks," in *Proceedings of the 21st international conference on pattern recognition (ICPR2012)*. IEEE, 2012, pp. 3304–3308.

[66] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at Microsoft," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8604–8608, doi: 10.1109/ICASSP.2013.6639345

[67] M. Nickel and D. Kiela, "Poincaré embeddings for learning hierarchical representations," 2017, arXiv:1705.08039

[68] B. P. Chamberlain, J. Clough, and M. P. Deisenroth, "Neural embeddings of graphs in hyperbolic space," preprint v2, 2017, arXiv:1705.10359

[69] M. M. Bronstein, J. Bruna, T. Cohen, and P. Velickovic, "Geometric deep learning: grids, groups, graphs, geodesics, and gauges," preprint v2, 2021, arXiv:2104.13478

[70] T. S. Cohen and M. Welling, "Steerable CNNs," 2016, arXiv:1612.08498

[71] T. S. Cohen, M. Geiger, J. Koehler, and M. Welling, "Spherical CNNs," 2018, arXiv:1801.10130

[72] D. P. Kingma and J. L. Ba, "ADAM: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015, arXiv:1412.6980

[73] K. Batzner, L. Heckler, and R. König, "EfficientAD: Accurate visual anomaly detection at millisecond-level latencies," 2024, arXiv:2303.14535

[74] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018, arXiv:1710.10903

[75] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," preprint v4, 2016, arXiv:1609.02907

[76] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023, arXiv:1706.03762

[77] T. N. Kipf and M. Welling, "Variational graph auto-encoders," 2016, arXiv:01611.07308

[78] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017, arXiv:1703.06103

[79] IBM, "What is a neural network?" 2024, accessed: 2024 May 22. url: https://www.ibm.com/topics/neural-networks.

[80] Z.-H. Zhou, *Machine learning*, ser. Springer eBook Collection. Singapore: Springer, 2021, doi: 10.1007/978-981-15-1967-3

[81] IBM, "What is deep learning?" 2024, accessed: 2024 May 22. url: https://www.ibm.com/topics/deep-learning.

[82] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022, doi: 10.1109/TNNLS.2021.3084827

[83] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791

[84] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018, pp. 4438–4445, doi: 10.1609/aaai.v32i1.11782

[85] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018, arXiv:1706.02216

[86] Y. Ma, S. Wang, C. C. Aggarwal, and J. Tang, "Graph convolutional networks with EigenPooling," preprint v2, 2019, arXiv:1904.13107

[87] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," 2019, arXiv:1806.08804

[88] A. Einstein and L. Infeld, *The evolution of physics: the growth of ideas from early concepts to relativity and quanta*. New York: Simon and Schuster, 1938.

[89] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," preprint v2, 2015, arXiv:1509.09292

[90] F. Rousseau, E. Kiagias, and M. Vazirgiannis, "Text categorization as a graph classification problem," in *Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 2015, pp. 1702–1712, doi: 10.3115/v1/P15-1164

[91] E. Q. Effah, E. O. Osei, M. D. Jnr., and A. Tetteh, "Hybrid approach to classification of ddos attacks on a computer network infrastructure," vol. 17, p. 19–43, Feb. 2024, doi: 10.9734/ajrcos/2024/v17i4428

[92] A. Hashemi and A. H. Pilevar, "Mass detection in lung CT images by using graph classification," *International Journal of Electronics and Electrical Engineering*, vol. 3, no. 3, 2013, url: https://vixra.org/abs/1405.0081.

[93] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A*, April 13, 2016, doi: 10.1098/rsta.2015.0202

[94] J. Bisgard, *Analysis and Linear Algebra: The Singular Value Decomposition and Applications*. American Mathematical Society, June 2021.

[95] M. Goshtasbpour, "Singular value decomposition, hessian errors, and linear algebra of non-parametric extraction of partons from DIS," 2014, arXiv:1408.6113

[96] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[97] P. Boldi and S. Vigna, "Axioms for centrality," preprint v2, 2013, arXiv:1308.2140

[98] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977, doi: 10.2307/3033543

[99] P. Bonacich, "Power and centrality: a family of measures," *American Journal of Sociology*, vol. 92, no. 5, pp. 1170–1182, 1987, doi: 10.1086/228631

[100] D. Stevanović, *Spectral Radius of Graphs*. Cham: Springer International Publishing, 2018, pp. 83–130, doi: 10.1007/978-3-319-70953-6_3

[101] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and Leman go neural: higher-order graph neural networks," preprint v5, 2018, arXiv:1810.02244

[102] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," preprint v1, 2012, arXiv:1207.0580

[103] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," preprint v2, 2012, arXiv:1206.5533

[104] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, November 2016.

[105] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*. Springer, 2021, doi: 10.1007/978-1-0716-1418-1

[106] F. Chollet, *Deep Learning with Python*. Manning Publications, 2018.

[107] S. J. Simske, *Meta-Algorithmics: Patterns for Robust, Low Cost, High Quality Systems*. John Wiley & Sons, 2013.

[108] J. B. Grimbleby, "Automatic analogue network synthesis using genetic algorithms," in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, 1995, pp. 53–58, doi: 10.1049/cp:19951024

[109] A. Das and R. Vemuri, "An automated passive analog circuit synthesis framework using genetic algorithms," in *IEEE Computer Society Annual Symposium on VLSI*, 2007, pp. 145–152, doi: 10.1109/ISVLSI.2007.22

[110] G. Sussman and R. Stallman, "Heuristic techniques in computer-aided circuit analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, no. 11, pp. 857–865, 1975, doi: 10.1109/TCS.1975.1083985

[111] R. Harjani, R. Rutenbar, and L. Carley, "OASYS: a framework for analog circuit synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 12, pp. 1247–1266, 1989, doi: 10.1109/43.44506

[112] Z. A. Lomnicki, "Two-terminal series-parallel networks," *Advances in Applied Probability*, vol. 4, no. 1, pp. 109–150, 1972, doi: 10.2307/1425808

[113] Y. Isokawa, "Series-parallel circuits and continued fractions," *Applied Mathematical Sciences*, vol. 10, no. 27, pp. 1321–1331, 2016, doi: 10.12988/ams.2016.63103

[114] A. E. Bayrak, Y. Ren, and P. Y. Papalambros, "Topology generation for hybrid electric vehicle architecture design," *ASME Journal of Mechanical Design*, vol. 138, no. 8, p. 081401, 2016, doi: 10.1115/1.4033656

[115] J. M. del Castillo, "Enumeration of 1-DOF planetary gear train graphs based on functional constraints," *ASME Journal of Mechanical Design*, vol. 124, no. 4, pp. 723–732, 2002, doi: 10.1115/1.1514663

[116] W. Ma, A. Trusina, H. El-Samad, W. A. Lim, and C. Tang, "Defining network topologies that can achieve biochemical adaptation," *Cell*, vol. 138, no. 4, pp. 760–773, 2009, doi: 10.1016/j.cell.2009.06.013

[117] R. Buettner, D. R. Herber, P. Abolmoali, and S. S. Patnaik, "An automated design tool for the generation and selection of optimal aircraft thermal management system architectures," in *AIAA Propulsion and Energy 2021 Forum*, Aug. 2021, doi: 10.2514/6.2021-3718

[118] https://github.com/anthonysirico/GDL-for-Engineering-Design, commit: 0ade3c0.

Appendix A

Python Code

To follow is the Python code used for the iterative classification, the Principle Component Analysis, how the graph-based dataset was created, and the classification model used. The remainder of the code can be found in Ref. [118].

A.1 Iterative Classification Code

```
def run_iteration(iteration, iteration_dataset, path, hidden_channels,
 known_set_size, training_split, epochs):
    seed = np.random.randint(0, 1000)
    iteration_path = f'{path}'
    if not os.path.exists(iteration_path):
        os.makedirs(iteration_path)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = GCN(hidden_channels=hidden_channels).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    criterion = CrossEntropyLoss()
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='
       min', factor=0.7, patience=5, min_lr=0.00001)

    for run in range(1,6):
        random.seed(seed)
        np.random.seed(seed)
        torch.manual_seed(seed)

        run_path = f'{iteration_path}/run_{run}'
        if not os.path.exists(run_path):
            os.makedirs(run_path)

        if run == 1:
            N = len(iteration_dataset)
            known_set = random.sample(iteration_dataset, int(N *
               known_set_size))

            known_utility = [graph.obj.item() for graph in known_set]
            known_median_utility = np.median(known_utility)

            for graph in known_set:
                graph.it_y = torch.tensor([1]) if graph.obj <
                   known_median_utility else torch.tensor([0])

            known_set = CreateDataset(known_set)
            unknown_set = iteration_dataset[int(N * known_set_size):]

            training_set = known_set[:int(len(known_set) * training_split)]
            validation_set = known_set[int(len(known_set) * training_split):]
```

```
        training_loader = DataLoader(training_set, batch_size=32, shuffle=
            True)
        validation_loader = DataLoader(validation_set, batch_size=1,
            shuffle=True)
        unknown_loader = DataLoader(unknown_set, batch_size=1, shuffle=
            False)
    else:
        n_known_min = known_min
        n_known_ones = len(known_ones_graphs)
        if n_known_ones < n_known_min:
            n_known_needed = n_known_min - n_known_ones
            if n_known_needed > len(predicted_ones_graphs):
                print('Not␣enough␣predicted␣ones')
                break

            n_known_needed_index = random.sample(range(len(
                predicted_ones_graphs)), n_known_needed)
            n_known_needed_graphs = [predicted_ones_graphs[i] for i in
                n_known_needed_index]

            known_set = known_ones_graphs + n_known_needed_graphs
            unknown_set = [predicted_ones_graphs[i] for i in range(len(
                predicted_ones_graphs)) if i not in n_known_needed_index]

            known_utility = [graph.obj.item() for graph in known_set]
            known_median_utility = np.median(known_utility)

            for graph in known_set:
                graph.it_y = torch.tensor([1]) if graph.obj <
                    known_median_utility else torch.tensor([0])

            known_set_final = CreateDataset(known_set)
            known_set_final.shuffle()

            training_set = known_set_final[:int(len(known_set_final) *
                training_split)]
            validation_set = known_set_final[int(len(known_set_final) *
                training_split):]

            training_loader = DataLoader(training_set, batch_size=32,
                shuffle=True)
            validation_loader = DataLoader(validation_set, batch_size=1,
                shuffle=True)
            unknown_loader = DataLoader(unknown_set, batch_size=1, shuffle
                =False)
        else:
            known_set = known_ones_graphs
            known_utility = [graph.obj.item() for graph in known_set]
            known_median_utility = np.median(known_utility)

            for graph in known_set:
                graph.it_y = torch.tensor([1]) if graph.obj <
                    known_median_utility else torch.tensor([0])

            known_set_final = CreateDataset(known_set)
            known_set_final.shuffle()
```

```
            training_set = known_set_final[:int(len(known_set_final) *
                training_split)]
            validation_set = known_set_final[int(len(known_set_final) *
                training_split):]

            training_loader = DataLoader(training_set, batch_size=32,
                shuffle=True)
            validation_loader = DataLoader(validation_set, batch_size=1,
                shuffle=True)
            unknown_loader = DataLoader(unknown_set, batch_size=1, shuffle
                =False)

    validation_accuracy = []
    for epoch in tqdm(range(1, epochs + 1)):
        train(model, training_loader, criterion, optimizer, device)
        val_acc = test(model, validation_loader, device)
        validation_accuracy.append(val_acc)
        scheduler.step(val_acc)

    predictions, probabilities = predict(model, unknown_loader, device)

    unknown_y_true = [1 if graph.obj < known_median_utility else 0 for
        graph in unknown_set]

    torch.save(model.state_dict(), f'{run_path}/model.pt')

    predicted_ones_index = [i for i, pred in enumerate(predictions) if
        pred == 1]
    predicted_zeros_index = [i for i, pred in enumerate(predictions) if
        pred == 0]

    predicted_ones_graphs = [unknown_set[i] for i in predicted_ones_index]
    predicted_zeros_graphs = [unknown_set[i] for i in
        predicted_zeros_index]

    predicted_ones_utility = [predicted_ones_graphs[i].obj.item() for i in
         range(len(predicted_ones_graphs))]

    predicted_zeros_utility = [predicted_zeros_graphs[i].obj.item() for i
        in range(len(predicted_zeros_graphs))]

    known_ones_index = [i for i, graph in enumerate(known_set) if graph.
        it_y.item() == 1]
    known_zeros_index = [i for i, graph in enumerate(known_set) if graph.
        it_y.item() == 0]

    known_ones_graphs = [known_set[i] for i in known_ones_index]
    known_ones_utility = [known_ones_graphs[i].obj.item() for i in range(
        len(known_ones_graphs))]
```

A.2 Principle Component Analysis Code

```
from typing import Tuple, Any
import pandas as pd
```

```
import numpy as np
from pandas import DataFrame, Index
from torch_geometric.utils import to_dense_adj, to_networkx
import networkx as nx
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from multiprocessing import Pool


def get_spectral_radius(A):
    eigenvalues = np.linalg.eigvals(A)
    largest_eigenvalue = np.max(np.abs(eigenvalues))
    return largest_eigenvalue

def compute_graph_features(graph):
    avg_harmonic_centrality = np.average(graph.x[:, 1].tolist())
    graph_nx = to_networkx(graph)
    btwn_cent = np.average(list(nx.betweenness_centrality(graph_nx).values
        ()))
    eig_cent = np.average(list(nx.eigenvector_centrality(graph_nx,
        max_iter=10000).values()))
    utility = graph.obj.item()
    spectral_radius = get_spectral_radius(to_dense_adj(graph.edge_index).
        numpy())
    compiled = graph.compile.item()
    simulated = graph.simulate.item()
    return avg_harmonic_centrality, btwn_cent, eig_cent, utility,
        spectral_radius, compiled, simulated

class CreatePCA:
    def __init__(self, dataset, n_components):
        self.dataset = dataset
        self.n_components = n_components

    def create_features_dataframe(self) -> tuple[DataFrame, Any]:
        results = [compute_graph_features(item) for item in self.dataset]
        unpacked_results = list(zip(*results))
        self.avg_harmonic_centrality, self.btwn_cent, self.eig_cent, self.
            utility, self.spectral_radius, self.compiled, self.simulated =
             unpacked_results

        features = pd.DataFrame({
            'AHC': self.avg_harmonic_centrality,
            'BC': self.btwn_cent,
            'EC': self.eig_cent,
            'SR': self.spectral_radius,
            'COMP': self.compiled,
            'SIM': self.simulated
        })

        feature_names = features.keys()[0:4]
        features['COMP'].replace(0, 'Did Not Compile', inplace=True)
        features['COMP'].replace(1, 'Compiled', inplace=True)
        features['SIM'].replace(0, 'Did Not Simulate', inplace=True)
        features['SIM'].replace(1, 'Simulated', inplace=True)

        return features, feature_names
```

```
    def process_principle_components(self):

        features, names = self.create_features_dataframe()

        x = features.loc[:, names].values
        x = StandardScaler().fit_transform(x)

        feature_cols = ['feature'+str(i) for i in range(1, x.shape[1]+1)]

        normalized_features = pd.DataFrame(x, columns=feature_cols)

        pca_features = PCA(n_components=self.n_components)
        pc_features = pca_features.fit_transform(x)

        principle_components_cols = ['Principle Component'+str(i) for i in
            range(1, self.n_components+1)]
        principle_features_df = pd.DataFrame(data=pc_features, columns=
           principle_components_cols)

        return principle_features_df
```

A.3 Example Pytorch–Geometric Set Build

```
from abc import ABC
from tqdm import tqdm
import pandas as pd
import torch
from torch_geometric.data import InMemoryDataset
from torch_geometric.utils import from_networkx
import networkx as nx
from sklearn.preprocessing import LabelEncoder
from scipy.io import loadmat
import numpy as np
from PCA import CreatePCA


def _get_node_features(networkx_graph, encoded_node_features):
    harmonic = list(nx.harmonic_centrality(networkx_graph).values())
    features = dict(enumerate(zip(encoded_node_features, harmonic)))
    nx.set_node_attributes(networkx_graph, features, 'feature')
    return networkx_graph


def _get_graph_features(A):
    eigenvalues = np.linalg.eigvals(A)
    largest_eigenvalue = np.max(np.abs(eigenvalues))

    return torch.tensor(largest_eigenvalue, dtype=torch.float32)


class TMSDataset(InMemoryDataset, ABC):
    def __init__(self, root, transform=None, pre_transform=None,
       pre_filter=None):
```

```
        super(TMSDataset, self).__init__(root, transform, pre_transform,
            pre_filter)
        self.data, self.slices = torch.load(self.processed_paths[0])

    def raw_file_names(self):
        return ['latest_data.mat']

    def processed_file_names(self):
        return 'tms_graphs.pt'

    def download(self):
        pass

    def process(self):
        global y
        tms_graphs = []

        raw_data = loadmat(self.raw_paths[0], squeeze_me=True)
        graph_data = pd.DataFrame(raw_data['python_graph_data'])
        node_features = graph_data['nl']
        encoder = LabelEncoder()
        encoded_node_features = node_features.apply(encoder.fit_transform)

        for index in tqdm(range(len(graph_data))):
            nx_graph = nx.DiGraph(graph_data['A'][index])
            nx_graph = _get_node_features(nx_graph, encoded_node_features[
                index])
            pyg = from_networkx(nx_graph, group_node_attrs=['feature'])
            pyg.index = index
            pyg.compile = torch.tensor(graph_data['comp'][index], dtype=
                torch.long)
            pyg.simulate = torch.tensor(graph_data['sim'][index], dtype=
                torch.long)
            pyg.catalog = torch.tensor(graph_data['CatalogCodeIndex'][
                index], dtype=torch.long)
            pyg.obj = torch.tensor(graph_data['bestObj'][index], dtype=
                torch.float32)

            tms_graphs.append(pyg)

        principle_components = CreatePCA(tms_graphs, 2)
        principle_components = principle_components.
            process_principle_components()

        for i in range(len(tms_graphs)):
            tms_graphs[i].graph_features = torch.tensor(
                principle_components.iloc[i].values,
                                                        dtype=torch.
                                                            float32).
                                                            unsqueeze(0)
            if graph_data['comp'][i] == 0:
                tms_graphs[i].graph_label = torch.tensor([1, 0, 1, 0],
                    dtype=torch.float).reshape(1, -1)
            elif graph_data['comp'][i] == 1 and graph_data['sim'][i] == 0:
                tms_graphs[i].graph_label = torch.tensor([0, 1, 1, 0],
                    dtype=torch.float).reshape(1, -1)
            elif graph_data['comp'][i] == 1 and graph_data['sim'][i] == 1:
```

```
                tms_graphs[i].graph_label = torch.tensor([0, 1, 0, 1],
                    dtype=torch.float).reshape(1, -1)

            print(f'Graph␣{i}␣has␣label␣{tms_graphs[i].graph_label}')

        if self.pre_filter is not None:
            tms_graphs = [graph for graph in tms_graphs if self.pre_filter
                (graph)]

        if self.pre_transform is not None:
            tms_graphs = [self.pre_transform(graph) for graph in
                tms_graphs]

        torch.save(self.collate(tms_graphs), self.processed_paths[0])
```

A.4 The Model

```
import torch
from torch_geometric.nn import GraphConv, global_mean_pool
from torch.nn import Linear
import torch.nn.functional as F

class GCN(torch.nn.Module):
    def __init__(self, hidden_channels, num_node_features=2,
        output_channels=2):
        super(GCN, self).__init__()

        self.conv1 = GraphConv(num_node_features, hidden_channels)
        self.conv2 = GraphConv(hidden_channels, hidden_channels)
        self.conv3 = GraphConv(hidden_channels, hidden_channels)
        self.lin = Linear(1026, output_channels)

    def forward(self, x, edge_index, batch, graph_features):
        x = F.relu(self.conv1(x, edge_index))
        x = F.relu(self.conv2(x, edge_index))
        x = self.conv3(x, edge_index)
        x = global_mean_pool(x, batch)

        x = torch.cat([x, graph_features], dim=-1)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin(x)

        return x
```

Appendix B

Python Modules

Table B.1: Python modules used in the case studies

Name	Version	Description
Python	3.10	An interpreted, object-oriented, high-level programming language with dynamic semantics. https://www.python.org
NumPy	1.24.2	An open-source project that enables numerical computing with Python. https://numpy.org
Pandas	1.5.3	An open source data analysis and manipulation tool, built on top of the Python programming language. https://pandas.pydata.org
Scipy	1.11.3	Provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics, and many other classes of problems. https://scipy.org
Pytorch	2.1.0	A Fully featured framework for building deep learning modules. https://pytorch.org
Pytorch Geometric	2.4.0	Consists of various methods for deep learning on graphs and other irregular structures. https://pytorch-geometric.readthedocs.io/en/latest/
SciKit Learn	1.3.1	An open-source machine learning library that supports supervised and unsupervised learning. https://scikit-learn.org/stable/
Concurrent		Provides a high-level interface for asynchronously executing callables.
Networkx	3.1	A Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks https://networkx.org/