A Graph Machine Learning Framework to Compute Zero Forcing Sets in Graphs

Obaid Ullah Ahmad, Mudassir Shabbir, Waseem Abbas, *Member, IEEE*, and Xenofon Koutsoukos, *Fellow, IEEE*

***Abstract*—This article studies the problem of computing zero-forcing sets (ZFS) in graphs and provides a machine-learning solution. Zero-forcing is a vertex coloring process to color the entire vertex set from a small subset of initially colored vertices constituting a ZFS. Such sets have several applications in network science and networked control systems. However, computing a minimum ZFS is an NP-hard problem, and popular heuristics encounter scalability issues. We investigate the greedy heuristic for this problem and propose a combination of the random selection and greedy algorithm called the random-greedy algorithm, which offers an efficient solution to the ZFS problem. Moreover, we enhance this approach by incorporating a data-driven solution based on graph convolutional networks (GCNs), leveraging a random selection process. Our machine-learning architecture, designed to imitate the greedy algorithm, achieves significant speed improvements, surpassing the computational efficiency of the greedy algorithm by several orders of magnitude. We perform thorough numerical evaluations to demonstrate that the proposed approach is considerably efficient, scalable to graphs about ten times larger than those used in training, and generalizable to several different families of synthetic and real-world graphs with comparable and sometimes better results in terms of the size of ZFS. We also curate a comprehensive database comprising synthetic and real-world graph datasets, including approximate and optimal ZFS solutions. This database serves as a benchmark for training machine-learning models and provides valuable resources for further research and evaluation in this problem domain. Our findings showcase the effectiveness of the proposed machine-learning solution and advance the state-of-the-art in solving the ZFS problem.**

***Index Terms*—Zero-forcing Set, graph convolutional network, network controllability, leader selection problem.**

I. Introduction

DYNAMIC coloring in graphs is a process of coloring vertices iteratively according to some pre-defined rules and conditions. Based on the coloring rules, several variants of such colorings have been considered with many applications in network science and engineering, such as air traffic flow management [1], nucleic acid sequence design in biochemical networks [2], channel assignment in wireless networks [3], and community detection in social networks [4]. In addition, graph coloring also serves as an effective tool in solving other significant graph theory problems, for instance, graph partitioning [5] and clique computation [6].

Zero forcing is a dynamic coloring of vertices, which are initially colored either black or white. A black vertex can change the color of its white neighbor to black under some conditions (i.e., the black vertex has exactly one white neighbor). The goal is to select the minimum number of vertices in a graph, which, if colored black initially, will render all vertices black at the end of the coloring process. Such a subset of initial black vertices is called a *zero forcing set (ZFS)* in a graph (as explained in Section II-A). Zero forcing has applications in modeling various physical phenomena, including logic circuits analysis, disease spread analysis, and information spread in social networks [7], [8], [9], [10]. In particular, ZFS is an important notion in studying *network controllability*, which is a central phenomenon in the control of networked systems. Network controllability is manipulating a network of agents as desired by injecting external control inputs through a subset of agents called leaders. The crucial problem is determining the minimum leader agents to make the network completely controllable. It turns out that ZFS in a network characterizes a leader set for the network's complete controllability [10], [11] (as discussed in Section II-B).

Unfortunately, finding a minimum ZFS for a given graph is a combinatorial optimization problem shown to be NP-Hard [12]. The algorithms to compute the optimal solutions take exponential time and are not scalable to huge networks. Algorithms computing approximate solutions, such as greedy, generally provide reasonable results; however, they also incur scalability problems as the network sizes grow. Additionally, degenerate cases exist for which the greedy solution can be arbitrarily bad [13]. We discuss the exact and approximate solutions in Section III.

This work aims to find a computationally fast and accurate solution for the ZFS problem using random selection, machine learning (ML) and data-driven approaches. Recently, ML-based solutions have found applications in solving computationally hard problems. Gama et al. propose a GNN-based distributed solution to solve the flocking and the multi-agent path planning problems [14]. They propose a novel framework using

Manuscript received 15 June 2023; revised 27 October 2023; accepted 22 November 2023. Date of publication 4 December 2023; date of current version 23 February 2024. This work was supported by the National Science Foundation under Grants 2325416 and 2325417. Recommended for acceptance by Prof. X. Li. *(Corresponding author: Obaid Ullah Ahmad.)*

Obaid Ullah Ahmad is with the Electrical Engineering Department, The University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: Obaidullah.Ahmad@utdallas.edu).

Waseem Abbas is with the Systems Engineering Department, The University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: waseem.abbas@utdallas.edu).

Mudassir Shabbir and Xenofon Koutsoukos are with the Computer Science Department, Vanderbilt University, Nashville, TN 37235 USA (e-mail: mudassir.shabbir@vanderbilt.edu; xenofon.koutsoukos@vanderbilt.edu).

Digital Object Identifier 10.1109/TNSE.2023.3337750

graph signal processing (GSP) to learn the controllers for these problems. Additionally, there has been an increasing interest in utilizing these data-driven, and graph machine learning (GML) approaches to solve hard combinatorial optimization problems [15], [16]. Cappart et al. provide a comprehensive review of recent works involving the use of machine learning to solve combinatorial optimization problems [16]. By exploiting the known instances' solution patterns and relating them to the underlying network structure, we can train ML models that provide approximately optimal solutions to large networks in a fraction of the time than the best-known heuristic algorithms. Further, when combined with heuristic components, the learning models could compute solutions with much better approximations. Nevertheless, designing efficient and stable machine learning architectures for combinatorial problems has inherent challenges. Some significant challenges include the availability of datasets for learning, designing appropriate ML architectures, and scalability and transferability [16], [17], [18] (as discussed in detail in Section III-C).

Our approach relies on combining data-driven and algorithmic insights to find near-optimal ZFS. We study the structural aspects of the problem to design a graph convolutional network (GCN). We also generate a huge database using synthetic and real-world graph datasets to train our GCN model. As a result, we achieve comparable results to the greedy algorithm in a fraction of the time for huge networks. Our main contributions are:

- We propose a GCN-based architecture using the insights from the greedy algorithm. The key aspect is to design a GCN capable of learning to imitate the steps of the greedy algorithm much more efficiently. By employing this domain knowledge in the GCN designing process, we achieve a scalable, generalizable, and time-efficient solution, as explained in Section VI-D. We analyze the proposed GNN in detail and discuss various model parameters to obtain superior solutions in terms of time complexity and ZFS size.
- We study the greedy algorithm for the ZFS problem in detail and uncover some valuable insights into its underlying structure. In particular, we develop a hypothesis pertaining to the random selection of ZFS in the initial iterations of the greedy algorithm and empirically validate it across several distinct datasets. To the best of our knowledge, existing research has not explored the integration of random and greedy vertex selection strategies as a means of designing effective heuristics for combinatorial optimization problems, such as ZFS.
- We generate a huge database for the ZFS problem. We compute optimal solutions for synthetic graphs and greedy solutions for three synthetic and four real-world graph datasets.
- Using the dataset with the optimal solutions, we show that for the ZFS problem, the optimal data is not vital for the training of our proposed architecture to achieve satisfactory results. Instead, training can be done using approximate solutions. See Section VII for further details.
- We thoroughly evaluate the proposed solution on an extensive collection of graphs and show that our model is highly scalable. Furthermore, we train the GCN on smaller graphs and evaluate it on graphs ten times the sizes of graphs in the training set and observe a remarkable difference in computation time from the greedy algorithm.

ZFS computation is a hard problem, as explained in Section III. There are ways to find an approximate solution that are generally practicable but can have unrealistic time complexity for huge graphs. ML methods afford the great potential to provide time-efficient solutions to challenging combinatorial optimization problems. However, in literature, these solutions are not trivial. For instance, Joshi et al. show that for the Travelling salesman problem (TSP), the trained model's learning is usually limited to a certain scale level, and the model's performance drops drastically when evaluated on larger networks [19]. Researchers are trying to generalize machine learning solutions for the more extensive networks for the TSP [20], [21], [22]. Bello et al. introduce a neural network-based framework for solving combinatorial optimization, with a focus on the traveling salesman problem (TSP). It trains a recurrent neural network to predict city permutations, optimizing parameters using reinforcement learning and comparing training methods using negative tour length rewards [20]. Additionally, collecting optimal data for training is also a massive problem for hard combinatorial optimization problems [23]. We have devised a way to use machine learning without needing an optimal labeled dataset.

The article is organized as follows: Section II introduces the notations and definitions; Section III discusses known ways to compute ZFS and the potential and challenges of ML; Section IV describes the datasets; Section V discusses the structural aspect of the problem. In the last three sections, we propose a GCN-based architecture for the ZFS problem and evaluate several of its aspects including generalizability and scalability. Finally, Section IX concludes the article.

II. Preliminaries

An undirected graph $G = (V, E)$ models a multiagent network. Vertex set V and edge set $E \subset V \times V$ represent agents and interactions between them, respectively. The edge between vertices u and v is denoted by an unordered pair (u, v). The *neighborhood* of u is the set $\mathcal{N}_u = \{v \in V : (u, v) \in E\}$ and the *degree* of u is $\deg(u) = |\mathcal{N}_u|$. The *distance* between vertices u and v, denoted by $d(u, v)$, is the number of edges in the shortest path between them. Next, we define the zero forcing process and the respective terms and then its applications.

A. Zero Forcing Process

Definition (Zero forcing (ZF) Process): Consider a graph $G = (V, E)$, such that each $v \in V$ is colored either *black* or *white* initially. The ZF process is to iteratively change the colors of vertices using the following rule until no further changes are possible.

Color change rule: If $v \in V$ is colored black and has exactly one white neighbor u, change color of u to black.

We say that v *infected* u if the color of white vertex u is changed to black by any black vertex v.

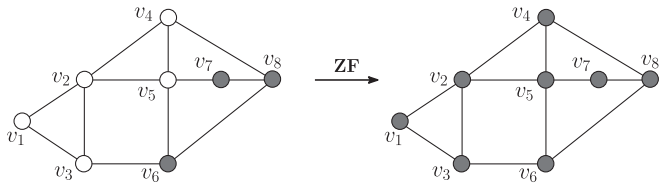


Fig. 1. $V' = \{v_6, v_7, v_8\}$ is the input set. After the ZF process, $D(V') = V$, as indicated by the black vertices. Hence, V' is a ZFS.

Definition (Derived Set): Consider a graph $G = (V, E)$ with $V' \subseteq V$ be the set of initial black vertices. Then, the set of black vertices obtained at the end of the ZF process is the derived set, denoted by $D(G, V')$, or simply $D(V')$ when the context is clear. The cardinality of a derived set for an input set V', i.e., $|D(V')|$, is the *span* of V'.

The set of initial black vertices V' is also referred to as the *input set*. For a given input set V', the derived set $D(V')$ is unique [24]. Now, we define the zero forcing set.

Definition (Zero Forcing Set (ZFS)): For a graph $G = (V, E)$, $V' \subseteq V$ is a ZFS if and only if $D(G, V') = V$. We denote a ZFS of G by $Z(G)$ and its size by $\zeta(G)$. The *minimum zero forcing set*, denoted as $\mathcal{Z}^*(G)$, is the ZFS of the minimum size.

Fig. 1 illustrates zero forcing and derived sets.

B. Applications of ZFS to Network Control

The zero forcing problem has many real-world applications. We are going to particularly talk about an application of ZFS in network controllability. We consider a network $G = (V, E)$ with $|V|$ agents, denoted by $V = \{v_1, v_2, \ldots, v_{|V|}\}$, of which m are leaders, which are represented by $V_L = \{\ell_1, \ell_2, \ldots, \ell_m\} \subseteq V$. We consider the following linear time-invariant system on G.

$$\dot{x}(t) = Mx(t) + Bu(t). \tag{1}$$

Here, $x(t) \in \mathbb{R}^{|V|}$ is the state vector and $u(t) \in \mathbb{R}^m$ is the external input injected into the system through m leaders. $M \in \mathcal{M}(G)$ is the system matrix, where $\mathcal{M}(G)$ is a family of symmetric matrices associated with G defined as

$$\mathcal{M}(G) = \{M \in \mathbb{R}^{|V|\times|V|} \ : \ M = M^\top, \\ \text{and for } i \neq j, M_{ij} \neq 0 \Leftrightarrow (i, j) \in E(G)\}.$$

The matrix $B \in \mathbb{R}^{|V|\times m}$ in (1) is the input matrix, such that $[B]_{ij} = 1$, if $v_i = \ell_j$; and 0 otherwise. We note that the input matrix B is defined by the selection of leader agents. Moreover, $\mathcal{M}(G)$ contains a broad class of system matrices defined on graphs, including the adjacency, Laplacian, and signless Laplacian matrices.

The system (1) is *controllable* if there exists an input $u(t)$ that can drive the system from an arbitrary initial state $x(t_0)$ to any desired state $x(t_f)$ in a finite amount of time. If the system is controllable for given system and input matrices, we say that (M, B) is a *controllable pair*. Moreover, (M, B) is a controllable pair if and only if the controllability matrix $\mathcal{C}(M, B) \in \mathbb{R}^{|V|\times|V|m}$ is full rank, i.e., $\text{rank}(\mathcal{C}(M, B)) = |V|$. The controllability matrix is defined as

$$\mathcal{C}(M, B) = \begin{bmatrix} B & MB & M^2B & \cdots & M^{|V|-1}B \end{bmatrix}.$$

Definition (Strong Structural Controllability (SSC)): A graph $G = (V, E)$ with a given set of leaders $V_L \subseteq V$ (and the corresponding B matrix) is *strong structurally controllable* if and only if (M, B) is a controllable pair for all $M \in \mathcal{M}(G)$.

We are interested in finding a minimum-sized leader set V_L that renders the network strong structurally controllable and refer to it as the *minimum leader selection problem (LSP)*. The LSP is computationally challenging and known to be NP-hard [12], [25], [26], [27]. The LSP is equivalent to the combinatorial optimization problem, called the *minimum zero forcing set (ZFS) problem*, of finding the *minimum ZFS* $\mathcal{Z}^*(G)$ of a given graph G. In [10], authors show that a leader set V_L renders the network strong structurally controllable if and only if V_L is a zero forcing set of the network graph G. Thus a minimum ZFS of G is also a solution of the LSP.

The ZFS problem is directly related to many other graph theoretic problems including the forts in graphs, the art gallery, and k-power propagation in hyper-graphs problems, etc. With respect to zero forcing, a fort is the set of vertices that do not get forced by some initial set of colored vertices, and a zero forcing set is exactly a set of vertices hitting every fort [28]. The number of distinct paths, the number of vertices in the paths, and the allowable multiplicities of a matrix for the graph are also shown to have a relation with the partial zero-forcing sets [29]. Similarly, there are a lot of other problems that can be indirectly solved using the zero-forcing set problem [27], [30], [31], [32].

III. Related Work

The computation of the *minimum ZFS* $\mathcal{Z}^*(G)$ is an NP-Hard problem [12]. Therefore, many researchers have looked into the computation of the approximate solution for this problem. In this Section, we mention some of the approaches for the exact computation of the miminum sized ZFS. We also discuss a few state-of-the-art algorithms to find approximate solutions, their shortcomings, and how machine learning can be utilized for the combinatorial optimization problems such as ZFS problem, along with its major challenges.

A. Exact ZFS Computation

In general, the computation of $\zeta(G)$ is an NP-hard problem [12]. Consequently, finding an optimal $Z(G)$ in polynomial time, in general, is out of reach. A widely used algorithm for the exact computation of $\zeta(G)$ and a minimum $Z(G)$ is a combinatorial algorithm called the *wavefront algorithm* [8], [33]. However, the algorithm has an exponential time complexity, and in the worst case, it is the same as enumerating all possible subsets of vertices [8]. Some improvements based on the integer programming formulation, branch-and-bound techniques, Boolean satisfiability models, bilevel mixed-integer linear program formulation have been proposed to solve the minimum ZFS problem [8], [34], [35]. However, the performance of such algorithms depends on several factors, including the existence of specific subgraphs and the graph density. Though these methods

offer improvements, their time complexity remains exponential in general. Hence, many researchers have proposed approximate algorithms to find the $Z(G)$ in polynomial time. In the following subsection, we discuss a few approaches to compute the approximate solutions for the minimum ZFS problem.

B. Approximate ZFS Computation

Various heuristics have been proposed to compute $Z(G)$ in large graphs in feasible times. The most notable is the *greedy heuristic* that typically computes a small-sized $Z(G)$. The main idea is to iteratively construct a $Z(G)$ by adding a vertex to a $Z(G)$ solution that maximizes the size of the derived set in that iteration. Once a solution is obtained by this iterative process, redundant vertices are removed to obtain a minimal $Z(G)$. The greedy heuristic generally performs well. Moreover, though the greedy heuristic is faster than optimal algorithms and it can be verified in linear time whether a given set is a $Z(G)$ [36], it still takes significant time to compute a solution for huge graphs as we demonstrate in our numerical evaluations (Section VI-D). Utilizing the idea of potential games, [13] presents another heuristic based on the *log-linear learning* (LLL) that returns comparable solutions (in terms of $\zeta(G)$) to the greedy algorithm. However, the quality of the solution is a function of the number of iterations. Numerical evaluations show a fast convergence; however, the evaluation is confined to graphs of small sizes (50 vertices). Some other heuristics are also discussed in [8], [34].

As mentioned above, the best-known heuristic to find the approximate solution for the ZFS problem is the greedy algorithm. However, it is not scalable to large graphs and has a huge time complexity for large graphs. For instance, in one of our numerical evaluations, it takes the greedy algorithm about 156 minutes to find $Z(G)$ of a graph with 990 vertices. Similarly, [13], [34] shows that there exist graphs for which the greedy solution can be arbitrarily bad; that is, the difference between the optimal and greedy solutions can be arbitrarily large. This limits the applicability of the greedy algorithm for real-world network problems.

C. Graph Machine Learning – Proposed Approach for ZFS Computation

Graph datasets present several new challenges for traditional machine learning like input size variations, non-regular neighborhood structure, and vertex permutations. There are two main approaches for graph-structured data. *Graph embeddings* transform the input graphs to a fixed-sized low-dimensional intermediate representation that is then used in a classifier. In contrast, *Graph Neural Networks (GNNs)* are trained in an end-to-end manner, and graph structural information is used to construct the layout of the Neural Network.

GNNs generalize deep neural networks (DNNs) for graph-structured data. The main objective is to achieve vector representations of vertices/graphs in the low-dimensional space that encodes structural relationships in graphs in the corresponding vector representations. A GNN is then trained in an end-to-end manner against some appropriately designed loss function using stochastic optimization methods to adapt to the given data samples while optimizing the learning model parameters. GNNs exploit the patterns and invariances in the given data distribution, which help solve many problems on graph-structured data like toxicity prediction in chemical compounds, community detection in social networks [37], etc. GNNs have also been applied to challenging combinatorial optimization problems on graphs with some success. Satisfiability, Travelling Salesman, Knapsack, Minimum Vertex Cover, and Maximum Cut are a few of the problems that have been solved using GNNs [21], [22], [38]. This is an exciting avenue for complicated combinatorial problems that would otherwise either take an exceptional time even for an approximate solution or could not be computed with limited computational resources. This work shows a GNN-based approach can be used to compute $Z(G)$ with reliable accuracy and efficiency. The sizes of $Z(G)$ returned by our GNN solution are comparable to or smaller than the existing techniques while taking only a fraction of runtime as we thoroughly compare and evaluate in the later sections. However, we note that in general, several challenges need to be addressed to design an efficient GNN solver for a hard combinatorial optimization problem like the minimum $Z(G)$.

1) Limited Datasets for Training: Constructing a required size dataset suitable for training a machine learning model is usually the most critical and challenging step in designing a data-driven solution for a combinatorial optimization problem. This is also true for the minimum ZFS problem for which there is no known labeled dataset available for learning. Generating a large enough data set with optimal solutions for hard combinatorial problems, such as computing a minimum $Z(G)$, requires due diligence along with sufficient computational resources [16]. We generate a huge new labeled database consisting of several synthetic and real-world graph datasets (Section IV).

2) Scaling and Generlizability: A related issue of *scaling* also plagues many proposed solutions. Even when there are some datasets available for some graph problems, the instances in these datasets are of very small sizes. Consequently, the machine learning models that are trained on these instances can only reliably be tested on similar-size problems. An important problem that some of the previous works have tried to address is generalizability, i.e., training a model on small instances that perform well on large instances. Such works report limited success so far [17]. Along with being scalable and generalizable to larger sizes of problems, the learned models should also *extrapolate and transfer* well to different families of graphs. Designing a machine learning solution that performs well outside the support of the training distribution is a crucial research challenge [18]. In this article, we present an approach to find the minimum $Z(G)$ that is a combination of both data-driven and heuristic methods (Section VI). We perform several experiments where we are able to illustrate that our approach is capable of both scaling and generalizing since we exploit the intricate details of the greedy algorithm in the design of our method. As the major part of the solution comes from the machine learning model, our architecture is also much faster and computationally inexpensive than the greedy heuristic without compromising on the quality of the solution.

3) GNN Architectures: GNN architectures for combinatorial optimization problems are designed to maximally exploit the structural characteristics of the problem during the learning phase. Thus, to learn the problem structure and solution patterns from the data, the architecture is adjusted for the particular problem to accommodate its unique characteristics, for instance, vertex penalties and related constraints [15]. This also requires devising proper loss functions for the GNN. Unfortunately, these adjustments are not straightforward and are not typically translated from one problem to another. As a result, designing the most appropriate GNN architecture with a suitable loss function for the combinatorial problem is a tricky affair. Li et al. proposed a novel approach to solve combinatorial optimization problems that leverage both deep learning and classic heuristics [39]. However, a very recent work of Böther et al. shows that for problems like MIS which involve guided tree searches, the GNNs do not necessarily learn meaningful representations and propose to use reinforcement learning inspired by the classical solvers for combinatorial optimization problems [17]. Hence, we look deeper into the greedy heuristics to solve the ZFS to provide a scalable GNN-based solution. Our machine learning model, inspired by the greedy solution, solves this optimization problem in a greedy fashion. We introduce a novel loss function for the task and refine the intermediate solution obtained from GNN model using a heuristic algorithm (Section VI).

In this work, we provide a GNN-based solution to solve the ZFS problem that is capable of effectively handling all these challenges and can provide comparable results as those of the greedy algorithm. We use Graph Convolutional Network (GCN) to imitate the steps of the greedy algorithm and iteratively complete the $Z(G)$ much faster than the greedy algorithm. The empirical results show that *approximate solutions can be used to train* our proposed network and the optimal solutions are not required for training for this problem. Additionally, the model trained on approximate (greedy solutions) performs better than the greedy algorithm itself in many cases. This provides empirical evidence that our proposed architecture can be used to generate a new database for the ZFS problem.

Next, we present some insights about the greedy algorithm for the ZFS problem and our proposed data-driven architecture to find the approximate $Z(G)$. First, we present details of a huge database prepared as a part of this work. Then, we present interesting observations about the greedy algorithm. Then, we introduce a new GCN-based architecture motivated by the observations from the greedy algorithm to find the $Z(G)$ for a given graph. Lastly, we demonstrate the capabilities of the proposed architecture in terms of scalability and generalizability.

IV. Datasets

We employ a data-driven model in our proposed approach that requires sufficient labelled data for training. As mentioned in the previous section, the zero-forcing set problem is a part of the combinatorial optimization problems class that are NP-Hard and considerable computational resources are required to find its solution. The algorithms that provide the optimal $Z(G)$ for a given graph G take exponential time and their applicability is limited to graphs of small sizes. The greedy heuristics on the other hand, take polynomial amount of time and perform fairly well on large random graphs. The learning-based solutions work comparatively very fast and are proclaimed to perform well on combinatorial optimization problems since they may discover useful patterns in the data that are usually hard to specify by hand [21], [39]. To use these learning-based algorithms, a decent-sized labeled dataset with optimal $Z(G)$ is required. Unfortunately, there is no dataset for the minimum ZFS problem. In this work, besides proposing a learning-based solution, we also create a huge database that can be used for any learning-based algorithm to find minimal $Z(G)$. This database will be publicly available for the research community [1].

TABLE I
Datasets Details

Graph Dataset	# of Graphs	Range ($\|V\|$)	Avg ($\|V\|$)	Avg Degree	Avg $\zeta_{Gr}(G)$
small ER	979	$30-70$	65	12.86	40.56
Large ER	1500	$500-1000$	744.8	19.64	448.11
Scale-free	500	$50-1000$	316.5	2.73	168.94
COLLAB	5000	$32-492$	74.49	37.37	57.89
IMDB-BINARY	1000	$12-136$	19.7	8.89	16.13
IMDB-MULTI	1500	$7-89$	13.0	8.1	10.97
REDDIT-BINARY	1834	$6-1194$	288.1	2.34	177.7

In our database, we include synthetic as well as real-world graph datasets. The computation of optimal $Z(G)$ is infeasible on datasets with huge graphs. As the greedy algorithm finds solutions that are fairly close to the optimal values, we compute greedy solutions $Z_{gr}(G)$ of sizes denoted by $\zeta_{gr}(G)$ for all the graphs. The total time to find the greedy solutions for all the datasets is about *a couple of months* on a machine having Xeon(R) Gold 6238R CPU @ 2.20 GHz CPU and A40 PCIe GPU. We use the same machine for all our experiments. In addition to the greedy solutions, we also generate optimal solutions for a small subset of synthetic graphs for a thorough study of our proposed architecture presented in Section VII. Following are the details of the graph datasets used in our experiments.

A. Synthetic Datasets

We generate three new labelled datasets of synthetic graphs. The total time to find the greedy solutions for these graph datasets is approximately more than a month. The details of the parameters to generate these datasets are presented below and the general statistics including the average $\zeta_{gr}(G)$ are mentioned in Table I.

-*Large ER graphs:* We generate a dataset of 1500 random Erdős-Rényi (ER) undirected graphs using the networkx python library with the number of vertices ranging between 500 and 1000. The density parameter p of ER graphs is varied between 0.013 to 0.125. The average number of vertices in a graph is 745, and the average number of edges is 14748.

-*Small ER graphs:* We synthesize 979 ER graphs of sizes between 30 and 70 where the average graph size is 65. The graphs in this dataset are relatively more dense with p varying

[1] https://tinyurl.com/ZFS-datasets

from 0.2 to 0.7. Only for these small graphs, we also compute optimal solutions using the wavefront algorithm [8]. In fact, the graphs in this dataset are selected from a pool so that, the greedy algorithm performs badly on them compared to the optimal solution. The average difference in the size of the optimal solutions $\zeta_{op}(G)$ and greedy solutions $\zeta_{gr}(G)$ is 6.1 vertices.

-*scale-free graphs:* This dataset contains 500 sparse undirected scale-free graphs synthesized using the same python module. The average degree of graphs in this dataset is 2.72 with the number of vertices ranging from 50 to 1000.

B. Real-World Datasets

We also include four standard real-world graph datasets in our database.

-*IMDB-BINARY & IMDB-MULTI* are both movie collaboration datasets where the nodes are the actors/actresses and an edge is formed if the actors appear in the same movie [40].

-*REDDIT-BINARY* is a social networks dataset consisting of graphs that represent the online Reddit discussions [41]. A node is a user and an edge represents a reply to the comments.

-*COLLAB* is a scientific collaboration dataset in which graphs have researchers as nodes and their collaborations as edges [42].

Among these real-world datasets, REDDIT-BINARY is the only one that contains graphs with more than 500 nodes. For this dataset, we only take the graphs that have less than 1200 nodes since it becomes computationally very expensive to find the greedy solutions for larger graphs. The total time for computing greedy solutions for REDDIT-BINARY is approximately 6 days. The graphs in the other datasets are prepared within a day. Further statistics about these datasets are provided in Table I.

These datasets are vital for the numerical analysis of the ZFS problem. In the next section, we provide some intriguing observations about the greedy algorithm using this database.

V. Insights Into Greedy Heuristics for ZFS Computation

One of the approaches to compute approximate $Z(G)$ is the greedy approach in which we iteratively construct a $Z(G)$ by adding a vertex to $Z(G)$ that maximizes the span of the intermediate solution as mentioned in Algorithm 1. However, the selection of a vertex will be arbitrarily random if two or more vertices have the same span. In the initial few iterations, we observe that the greedy algorithm is likely to have multiple potential vertices to add to the intermediate solution if the degree of all the vertices is large enough. During these iterations, the construction of the intermediate $Z(G)$ solution by greedy algorithm might be as effective as the random selection. Based on this observation, we propose the following conjecture.

Conjecture 1: If half of the vertices of $Z_{gr}(G)$ are picked randomly and the rest of the approximate solution is completed by the greedy solution, then it is fairly the same as obtaining a solution completely by the greedy algorithm.

Algorithm 1: Greedy Algorithm for ZFS.

Input: $G = (V, E)$
Output: Zero forcing set Z
Initialize: $Z \leftarrow \{\}$
while $D(Z) \neq V$ **do**
  $v^* \leftarrow \underset{v_i \in V \setminus Z}{argmax} |D(Z \cup \{v_i\})|$
  $Z \leftarrow Z \cup \{v^*\}$
end while
for all $v_i \in Z$ **do** // removing redundancies
  if $D(Z \setminus \{v_i\}) = V$ **then**
    $Z \leftarrow Z \setminus \{v_i\}$
  end if
end for

We define this process of the combination of random selection along with greedy heuristics as a *Random-Greedy* algorithm. We empirically validate Conjecture 1 utilizing the database presented in Section IV. We perform experiments on all four real-world, large ER, and scale-free graph datasets. Only a subset of large ER graphs dataset (300 graphs) is selected due to the computation overhead for these large graphs. The details of the experimental setup and the results are mentioned in the following subsections.

A. Experimental Setup

For all these synthetic and real-world datasets, we compute a greedy solution represented as $Z_{gr}(G)$ for a graph $G = (V, E)$ and the size of this set is represented as $\zeta_{gr}(G)$. The greedy algorithm has two dominant parts; first to greedily populate $Z(G)$ until it is a complete solution, and then a greedy redundancy check to remove any extra nodes. The redundancy check is a separate part, and it is applied in all our experiments where the greedy algorithm is used to complete the solution.

To corroborate our conjecture, we introduce a random selection in the greedy algorithm. We provide an intermediate solution computed by randomly selecting a vertex set which is a fraction of $\zeta_{gr}(G)$ and then pass this partially formed solution to the greedy algorithm to obtain a $Z(G)$ primarily based on random selection. We represent the solution obtained by the *random-greedy algorithm* as $Z_{rdm}(G)$ and its size as $\zeta_{rdm}(G)$. For comparison, we already have the greedy solution and the $\zeta_{gr(G)}$ for each graph, we only fix the randomness fraction ratio τ to obtain the intermediate solution.

The performance of this approach is described by using the percentage deviation between the size of the $Z(G)$ by the greedy solution $\zeta_{gr}(G)$, and the size of the $Z(G)$ returned by this partial random selection $\zeta_{rdm}(G)$. We report the average of the deviation over all the graph instances in the test sets of different datasets. Formally,

$$\text{Dev}(\zeta_{rdm}) = \frac{1}{|\mathcal{K}|} \sum_{i=0}^{|\mathcal{K}|} \frac{\zeta_{rdm}(G_i) - \zeta_{gr}(G_i)}{\zeta_{gr}(G_i)} \times 100,$$

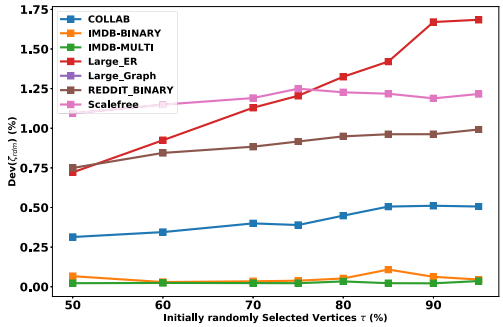


Fig. 2. Results for synthetic and real-world datasets when a certain percentage of vertices is selected randomly and the greedy algorithm is used to complete the $Z(G)$.

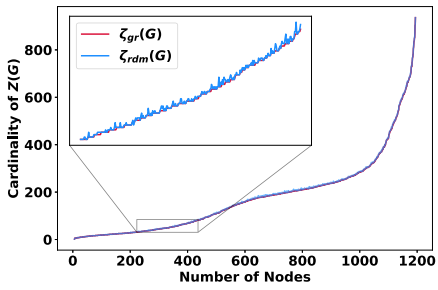


Fig. 3. Sizes of the $Z(G)$ computed by the greedy and the random selection for the graphs in the REDDIT BINARY dataset with $\tau = 0.7$.

where $\mathcal{K}$ is the test set of a dataset. We vary the fraction ratio τ for the random selection and observe the average deviation for each of the datasets.

B. Results and Discussion

The results obtained from this experiment align with Conjecture 1. In this experiment, we begin by randomly selecting a subset of vertices from a given graph G as part of the $Z_{rdm}(G)$ solution. The size of this subset is determined by multiplying a user-defined parameter, τ, by the size of the ZFS obtained from the greedy algorithm, denoted as $\zeta_{gr}(G)$. Subsequently, we feed this partial random solution to the greedy algorithm to acquire a complete solution. Finally, a redundancy check is performed to eliminate any redundant vertices from the solution $Z_{rdm}(G)$.

To evaluate the impact of the random selection, we vary the percentage of the random selection and plot the average deviation $Dev(\zeta_{rdm})$ from the sizes of the ZFS obtained by the greedy algorithm. Fig. 2 shows the deviation of ζ_{rdm} from ζ_{gr} for all the datasets under consideration. Notably, we observe that on average, the size of $Z_{rdm}(G)$ remains reasonably close to the size of $Z_{gr}(G)$, even when a considerable portion of $Z(G)$ is selected randomly. For instance, even if we pick $0.95 \times \zeta_{gr}(G)$ vertices randomly and then complete the rest of the solution using the greedy algorithm, then the solution $Z_{rdm}(G)$ merely has 1% more vertices on average than the purely greedy solution $Z_{gr}(G)$. This finding substantiates Conjecture 1. The increasing trend for the Large ER graphs dataset in Fig. 2 is because the graphs in the dataset are relatively sparser, and a slight difference in the size of $Z(G)$ results in a larger deviation from the greedy algorithm. Fig. 3 compares the sizes of $Z(G)$ for graphs in the REDDIT BINARY dataset with $\tau = 0.7$. It can be clearly observed that the results from the random selection are usually as small in size as the results of the greedy algorithm.

As explained above, the main reason behind this conjecture is that for the first several iterations, the greedy algorithm picks the vertices randomly but the later iterations are important in choosing the vertices that usually increase the size of the derived set significantly. Consider the example in Fig. 4. The optimal $Z(G)$ size for this graph is three. If we pick two vertices randomly and the third using the greedy approach, we can still end up with $Z(G)$ of size three. This is primarily because, for

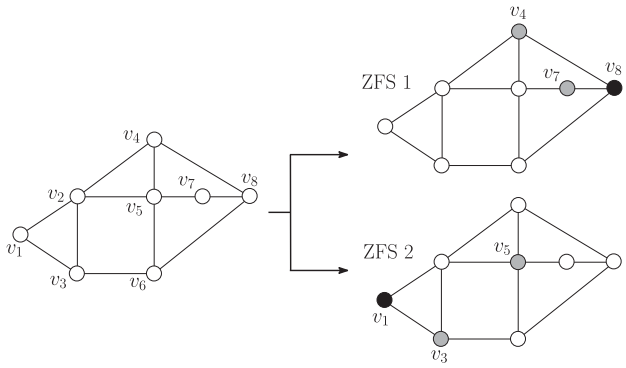


Fig. 4. Two instances of ZFS solution. The initial two vertices (v_4 and v_7 in ZFS 1, and v_3 and v_5 in ZFS 2) are selected randomly, whereas, the third vertex in each solution (v_8 in ZFS 1 and v_1 in ZFS 2) is selected by the greedy algorithm maximizing the span.

the first iteration, the derived set has only one vertex (itself), no matter which vertex is selected. Assume vertex v_7 is selected randomly (greedy can select this vertex as well). Now, whichever vertex is picked next can increase the size of the derived set by at most two. If v_4 is picked next by a random process, then the last vertex picked by the greedy algorithm will be v_8, which will complete the $Z(G)$. Similarly, if v_5 and v_3 are the first two randomly selected vertices, v_1 will be picked by the greedy algorithm to maximally increase and complete the $Z(G)$. This example illustrates that not only can the first iterations of the greedy algorithm be random, but there can also be multiple optimal solutions of $Z(G)$ for a given graph. Hence, a random process can arguably replace the greedy process for initial iterations. However, the random selection can result in a sub-optimal solution slightly larger than the original solution. The redundancy check, in those cases, removes the extra vertices from the solution and brings it closer to the optimal $Z(G)$ in size.

To further illustrate this point, we plot the size of the derived set after each iteration of the greedy algorithm for a few graphs in Fig. 5. These graphs are selected so that the difference between the size of the graph and $\zeta_{gr}(G)$ is significant. We plot the size of the derived set after the addition of each vertex by the greedy algorithm. It can be easily observed that there comes a point where by the addition of a very small subset of nodes, the span of the solution sees a drastic jump. For instance, for the graph from REDDIT-BINNARY dataset, we see that for the addition

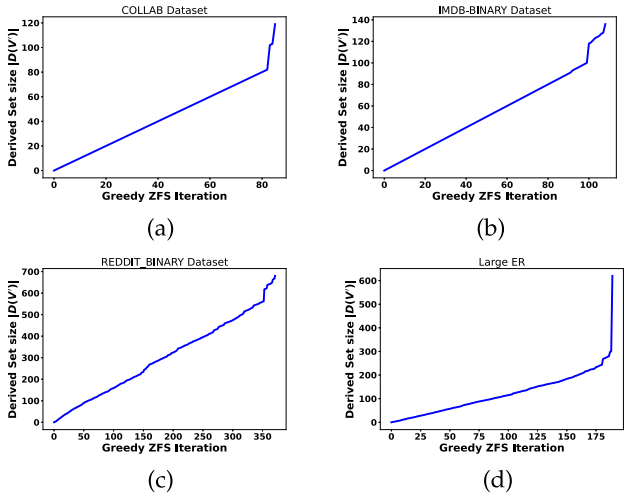


Fig. 5. Plots for size of derived set against each greedy algorithm iteration.

of the last 20 vertices, the size of the derived set grows by about 120 vertices. Similar behavior is observed from graphs of other datasets. This behavior explains the fact that only a fraction of iterations of the greedy heuristics, especially the later ones, are important.

In a nutshell, even if 95% of $Z(G)$ is populated by the random vertex selection and the rest is completed by the greedy algorithm, it can be seen that for all kinds of synthetic and real-world graphs, the average deviation is no more than 3%. Hence, *random selection along with greedy heuristics can reduce the $Z(G)$ computation time by manifolds without a significant loss in performance.* In the next section, we replace the greedy heuristics with a faster learning-based algorithm to further reduce the computation time.

VI. Graph Convolutional Network (GCN) Architecture for ZFS

Previously, we have shown that a sophisticated algorithm is required only for the last few iterations of the $Z(G)$ computation. In this section, we design a Graph Neural Network (GNN) based architecture to replace the greedy algorithm that can significantly reduce the time complexity from the greedy algorithm and can still generate a small-sized $Z(G)$. The proposed GNN is based on Graph Convolutional Network (GCN) [43] that uses a message-passing network where the information is communicated along the neighboring vertices within the graph. It works by iteratively aggregating information from neighboring vertices to update node representations, allowing the network to learn meaningful features from graph-structured data. It leverages the graph's connectivity patterns to improve its understanding of the relationships between vertices.

VI. Overview of the approach

Given a network G, our goal is to find a *binary labelling* for each vertex in G such that a label **1** represents that the vertex is included in a $Z(G)$, while label **0** means the converse. A machine learning approach to this problem involves training a model to generate this discrete labeling: a model $f(G)$ that takes an input graph G and outputs binary labeling of vertices. Generally, these models generate a probability map $P : V \rightarrow [0, 1]$ indicating how likely each vertex is to be included in a $Z(G)$. However, a straightforward application of this approach in which a vertex is included in a solution set based on the probability assignment does not always render a correct solution to the combinatorial optimization problem [39]. Having outputs that are not necessarily valid solutions is a general issue in data-driven approaches for combinatorial optimization problems, mainly because the training is performed to search the parameter space without any hard constraints. To deal with this issue, we train a separate *regression-based model* that predicts the size $\zeta_{op}(G)$ of an optimal solution using basic network properties. We obtain an *intermediate* solution $\hat{S}^x(G)$ by selecting $x = \mathcal{T}_{rand} \times \hat{\zeta}(G)$ vertices randomly from the vertex set V, where $\hat{\zeta}(G) \in \mathbb{Z}$ is the predicted value of $\zeta_{op}(G)$ obtained by the regression model, and $\mathcal{T}_{rand}$ is a hyper-parameter for the percentage of the solution selected by the random process. For simplicity, we drop the argument of $\hat{S}^x(G)$. We then pass the input graph G and this intermediate solution $\hat{S}^x$ as part of vertex features to the GCN to get the output probabilities for each vertex. These probabilities are sorted and the vertex with the maximum probability, which is already not a part of the intermediate solution, is added in $\hat{S}^x$ to form $\hat{S}^{x+1}$. We find the derived set $D(G, \hat{S}^{x+1})$ and check if $\hat{S}^{x+1}$ is a zero forcing set. If it is not a $Z(G)$, this partial solution is passed iteratively through GCN and a vertex is added to it in each iteration based on the GCN output probabilities until it becomes a $Z(G)$ denoted by $\hat{S}$. Fig. 6 illustrates this scheme. The GCN will iteratively add a vertex in each iteration to the partial solution until it becomes a ZFS. This is represented by the diamond-shaped block (named *ZFS check*) in Fig. 6. In the end, a redundancy check is applied to remove the extra vertices from the solution that might have been added as a result of the initial random selection or by the GCN, and we obtain a final solution $Z_{gcn}(G)$ of size $\zeta_{gcn}(G)$. Hence, the solution obtained from the proposed approach is essentially a ZFS.

In the following subsections, we briefly explain the main parts of the proposed architecture.

A. Regression Model

The regression model is a simple *Random Forest* model from the python module *sklearn*. The random forest is a meta estimator that fits a number of classifying decision trees on several samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting [44]. The number of estimating trees used in our experiments are 500 and the objective function for training the regression model is the squared error. The graph features used for training are the number of vertices in the graph G, number of edges, and the maximum five and the minimum six vertex degrees. This trained model predicts $\hat{\zeta}(G)$, the size of $\zeta(G)$. This model is computationally inexpensive but provides a good prediction with a very low error as mentioned in Section VI-D.

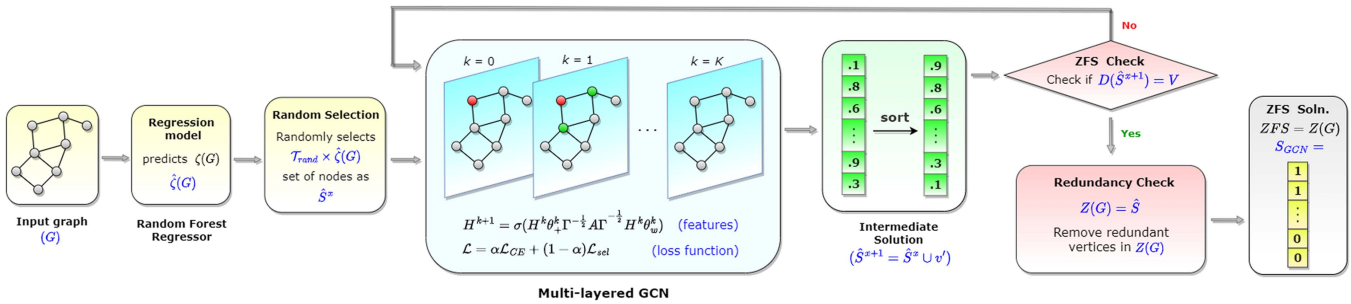


Fig. 6. Algorithm overview. First, a regression model trained on $\zeta(G)$ provides an estimate $\hat{\zeta}(G)$ of $\zeta(G)$. An intermediate solution $\hat{S}^x$ is formed by randomly selecting $\mathcal{T}_{rand} \times \hat{S}$. This intermediate solution is then completed iteratively using a multi-layered graph convolutional network $f(G;\theta)$ that provides a set of probabilities for each vertex in input graph G until a ZFS $\hat{S}$ is formed. Lastly, a redundancy check is performed that removes any redundant vertices.

B. Random Selection

As explained above, for the initial iterations, random selection can be as effective as greedy selection. Thus, we randomly populate the intermediate solution by a fraction of vertices. This fraction is computed using the $\hat{\zeta}(G)$ and the threshold $\mathcal{T}_{rand}$. For our experiments, we set $\mathcal{T}_{rand} = 0.5$. The output of this Random Selection process is a set of vertices $\hat{S}^x$ generated by randomly selecting $x = 0.5 \times \hat{\zeta}(G)$ number of vertices.

C. GCN

In our combinatorial optimization problem, the goal of the desired machine learning model is to learn a function $f(G;\boldsymbol{\theta}) : \mathcal{G} \to \{0,1\}^*$, where $\mathcal{G}$ is the set of all unlabeled graphs and $\{0,1\}^*$ is the space of all 0,1-vectors of arbitrary length depending on the size of graphs in $\mathcal{G}$. Being a data-driven approach, we use labeled data to train this *supervised* machine learning model. We assume that our data is in the form of pairs (G_i, S_i), where G_i is an undirected graph with N_i vertices and $S_i \in \{0,1\}^{N_i}$ is a binary representation of an optimal zero forcing set of G_i. As per standards, we let our approximation function $\hat{f}(G_i, \boldsymbol{\theta})$, a machine learning network, take values from $[0,1]^*$ instead of $\{0,1\}^*$ to facilitate a differentiable loss function. The network $\hat{f}(G_i;\boldsymbol{\theta})$ is parameterized by a learnable parameter $\boldsymbol{\theta}$, and is trained to predict S_i for a given G_i. The parameters (weights) of the network are updated by gradient descent while optimizing a loss function $\mathcal{L}(S_i, \hat{f}(G_i;\boldsymbol{\theta}))$ discussed in detail in Section VI-C1. The network $\hat{f}(G_i;\boldsymbol{\theta})$ is trained to imitate an iteration of the greedy algorithm. For training, we randomly remove a vertex from the ground truth ($Z_{gr}(G)$ in our case), and train the network to learn to predict that missing vertex. We initialize the first layer by the one-hot encoding of each vertex degree. We also concatenate to these vertex features a binary value indicating whether or not a vertex is a part of the intermediate solution $\hat{S}^x$. The features of a vertex v for each subsequent layer are denoted by H_v^{k+1}, where the superscript $(k+1)$ represents the layer index, and the subscript denotes the vertex. H_v^{k+1} is computed from layer-wise convolutions with previous layers by first aggregating the features of neighbors of v (including v itself). This so-called *message-passing* step precedes the convolution with the weight matrices $\boldsymbol{\theta}^k$ as explained in the (2). We get the updated feature vector by adding a bias parameter and applying a non-linearity to the resultant. Instead of using the adjacency matrix A of the input graph to aggregate neighborhood features, we use symmetric normalized matrix $\Gamma^{-\frac{1}{2}}A\Gamma^{-\frac{1}{2}}$ where Γ represents the diagonal matrix of the respective vertex degrees. Formally,

$$H^{k+1} = \sigma(H^k\boldsymbol{\theta}_b^k + \Gamma^{-\frac{1}{2}}A\Gamma^{-\frac{1}{2}}H^k\boldsymbol{\theta}_w^k), \tag{2}$$

where σ is any nonlinear activation function and $\boldsymbol{\theta}_b^k \in \mathbb{R}^{C^k \times 1}$ and $\boldsymbol{\theta}_w^k \in \mathbb{R}^{C^k \times C^{k+1}}$ are the trainable weights used for the convolutions. The C^k denotes the number of features per vertex at the kth layer.

The standard loss function used in GCN penalizes the wrong prediction of labeling of each vertex and aggregates the total loss over all the vertices. However, this may not be an ideal loss to minimize for the combinatorial optimization problem considered in this work. Therefore, we design a custom loss function suitable for learning effectively.

1) The Loss Function: A differentiable loss function is used in supervised learning to estimate the "distance" between the predicted label and ground truth and find the appropriate direction to update the neural network's weights using optimization methods like gradient descent. The loss function minimized during the training phase is conventionally the binary cross-entropy loss. For a training example G_i, S_i, the binary cross-entropy is defined as

$$\mathcal{L}_{CE}(S_i, \hat{f}(G_i;\boldsymbol{\theta})) = \sum_{j=1}^{N}(S_{ij}\log(\hat{f}_j(G_i;\boldsymbol{\theta})) + (1-S_{ij})\log(1-\hat{f}_j(G_i;\boldsymbol{\theta}))),$$

where S_{ij} is the jth element of S_i and $\hat{f}_j(G_i;\boldsymbol{\theta})$ is the jth element of $\hat{f}(G_i;\boldsymbol{\theta})$ (representing the jth vertex). This standard loss penalizes the wrong prediction of labeling of each vertex and aggregates the total loss over all the vertices. In addition to this loss, we add another term that further penalizes the vertices that are already a part of the intermediate solution. This term forces the model to learn to imitate the iterative steps of the greedy algorithm. We update our loss function as follows:

$$\mathcal{L}(S_i, f(G_i;\boldsymbol{\theta})) = \alpha\mathcal{L}_{CE}(S_i, f(G_i;\boldsymbol{\theta})) + (1-\alpha)\mathcal{L}_{sel}(\hat{S}_i^x, f(G_i;\boldsymbol{\theta})),$$

where $\mathcal{L}_{sel}(\hat{S}_i^x, f(G_i;\boldsymbol{\theta}))$ is the summation of the probabilities corresponding to the x vertices included in $\hat{S}_i^x(G)$. This function contains a hyper-parameter α that signifies the relative importance of penalizing the already selected vertices and can be used to tune the network. We set it to 0.5.

D. Evaluation of the Proposed Approach

To evaluate our proposed GCN based approach, we run experiments on all the available datasets mentioned in Section IV. We combine all the synthetic, and the real-world graph datasets and randomly split them into 90-10% train-test sets. Both the regression and the GCN models are trained on this 90% train set. The regression is trained to predict $\hat{\zeta}(G)$ whereas the GCN will predict the vertex that will be added to the partial solution $\hat{S}^x$ by the greedy algotithm in xth iteration.

The hyper-parameters of both the regression and GCN models are same throughout all the following sets of experiments. The Random Forest regression model has 500 estimators with a squared error loss function. This regression model is trained only once and is used for all the rest of the experiments of this article. Its mean absolute error and mean squared error for the test set are 3.42 and 6.87 respectively. On the other hand, the GCN model is trained separately for all the different sets of experiments. However, its hyper-parameters are kept the same. The main architecture of GCN is adapted from Li et al. [39]. There are 20 hidden layers, 500 vertex features for the first layer and 64 for all the consequent layers. For the first layer, we use one-hot encoded degree of each vertex where a maximum of 499° is feasible for training and testing. We also concatenate a binary indicator to inform the model whether or not a vertex is a part of $\hat{S}^x$. Each model is trained for 300 epochs with the custom loss function defined in Section VI-C1.

We evaluate our architecture on two main metrics: *accuracy*, and *time complexity*.

The accuracy is defined as the percentage deviation between the size of the $Z(G)$ by the greedy solution $\zeta_{gr}(G)$, and the size of the $Z(G)$ returned by GCN $\zeta_{gcn}(G)$. We report the average of deviation over all the graph instances in the test sets of different datasets. Formally, the average deviation of GCN solutions from greedy solutions is

$$\text{Dev}(\zeta_{gcn}) = \frac{1}{|\mathcal{K}|}\sum_{i=0}^{|\mathcal{K}|}\frac{\zeta_{gcn}(G_i) - \zeta_{gr}(G_i)}{\zeta_{gr}(G_i)} \times 100,$$

where $\mathcal{K}$ is the set of graphs in the test set. Fig. 7 compares $\zeta_{gr}(G)$, and $\zeta_{gcn}(G)$ for the graphs test dataset and the respective time to compute them. We have sorted the results on the basis of the size of the greedy solutions and the time taken by the greedy algorithm to compute the solution.

Despite having sparse as well as dense graphs in the test set with an average $\zeta_{gr}(G)$ to be 105.78, the average deviation $Dev(\zeta_{gcn})$ on this test set is 0.73%. This essentially means that the $Z(G)$ found by the proposed architecture $Z_{GCN}(G)$ on average has 0.76% more vertices than $Z_{gr}(G)$. However, for a single graph with 690 vertices, the GCN was able to find a $Z(G)$ containing 11 vertices (4.14% deviation) less than the greedy solution. On the other hand, there is a graph with 880 vertices for which the greedy solution had 28 vertices (5.9% deviation) less in its solution compared to the GCN solution. The total time taken by the greedy algorithm to find a solution for all the graphs in this test set is about 99 hrs whereas the proposed solution found all the solutions in approximately 8.2 hrs where most of the time is spent for redundancy check.

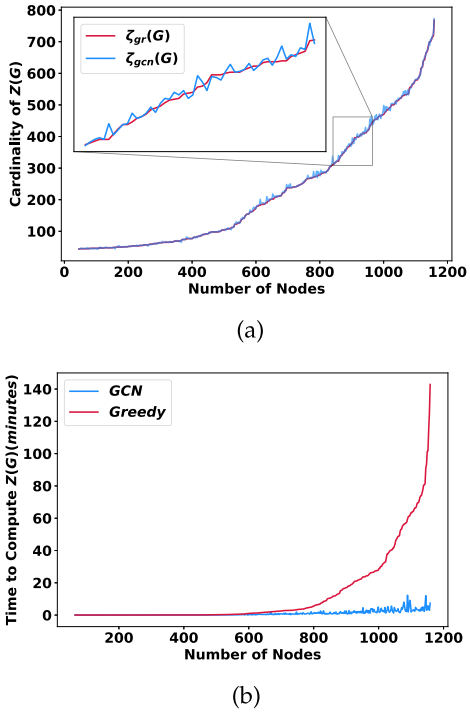


Fig. 7. Plot (a) shows the sizes of the ZFS computed by the greedy as well as the GCN-based solution for the graphs in the test set. Plot (b) shows the respective time taken to compute the solutions.

Evidently, the proposed architecture can find the $Z(G)$ 12 times faster than the greedy algorithm while being on average, only 1% larger in size. In the next section, we evaluate the learning capabilities of our architecture from sub-optimal solutions.

VII. Generalizability Over Sub-Optimal Solutions

Being a data driven approach, the proposed GCN based architecture requires optimal solutions for training. Unfortunately, these are usually computationally hard to compute with large enough graph instances needed for efficient learning. Thus, forming an optimal dataset is nearly impossible for NP-hard combinatorial problems. It is only natural to look for approximate solutions which are not very far from the optimal solutions and can be found in polynomial time. As verified by the numerical evaluations, our proposed architecture is capable of performing just as good whether it is trained on optimal or sub-optimal data. To validate this hypothesis, we use the *small ER graphs* dataset (explained in Section IV-A) for GCN training and testing since it is the only dataset for which the optimal solutions are available.

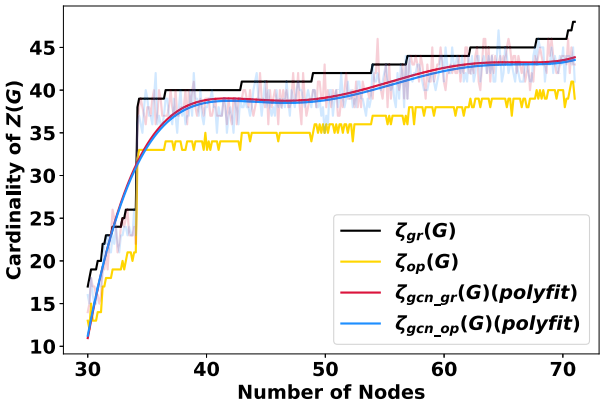
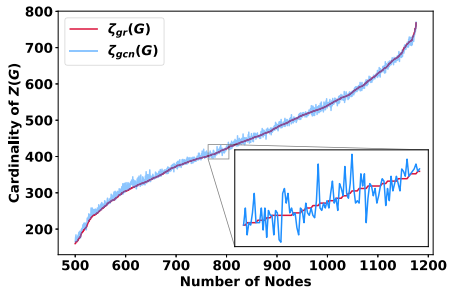


Fig. 8. Comparison of sub-optimal and optimal dataset training.

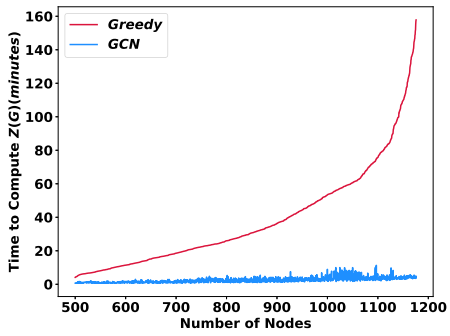
As far as the experimental setup is concerned to validate our hypothesis, only the training and testing sets are changed; the rest is the same as in Section VI-D. We randomly split the small ER graph dataset into $70-30\%$ train-test sets so that we have enough data to evaluate. Since we randomly remove a vertex from the greedy/optimal solution for training, we have enough samples for the training. We first train a GCN model on the optimal $Z(G)$ and use it to find the $Z(G)$ by our proposed approach. We call the output set $Z_{gcn\_op}(G)$ with size $\zeta_{gcn\_op}(G)$. Similarly, we train another GCN model using the greedy solutions as the ground truth. Since this model is trained on sub-optimal data, the $Z(G)$ obtained from our architecture is expected to be a little different from the one where the GCN is trained on the optimal data. The output from this GCN model is referred to as $Z_{gcn\_gr}(G)$ with size $\zeta_{gcn\_gr}(G)$. We plot the size of greedy solutions $\zeta_{gr}(G)$, optimal solutions $\zeta_{op}(G)$, the greedy-gcn solutions $\zeta_{gcn\_gr}(G)$, and the optimal-gcn solutions $\zeta_{gcn\_op}(G)$ in Fig. 8.

It can be observed from Fig. 8 that solutions from both the GCNs are fairly close to each other. In fact, the average deviation for both the models has a difference of 0.67%, and both perform better than the greedy algorithm. The model trained on the greedy solutions has the average deviation $Z_{gcn\_gr}(G)$ to be -5.13% from the greedy solutions whereas the model trained on the optimal solutions has the average deviation $Z_{gcn\_op}(G)$ to be -5.8%. This shows that both the GCN architectures will give a solution that will have, on average, 2 vertices less than the greedy solution. However, the difference between the two GCN solutions is negligible.

This experimental setup validates that *our proposed approach can learn equally well from the greedy as well as the optimal solutions*. Not only this, we are also able to show that GNNs can perform better than the greedy algorithms even when trained on the solutions from greedy heuristics. Hence, *using our proposed approach, optimal data is not necessary for the ZFS problem and the approximate solutions can be used for training.* This result paves a way for other combinatorial problems to be solved using GNNs without preparing an optimal dataset for training. In the next section, we validate the scalability of our approach and compare our results for huge graphs with the greedy algorithm.



(a)



(b)

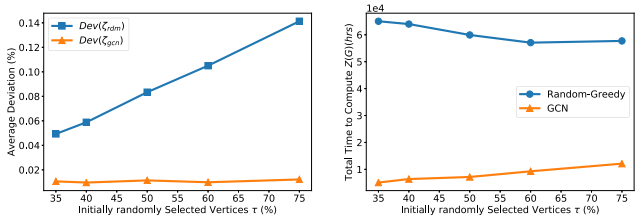
Fig. 9. Plot (a) shows the sizes of the $Z(G)$ computed by the greedy as well as the GCN-based solutions for the graphs in the test set of larger graphs. Plot (b) shows the respective time taken to compute the solutions.

VIII. Scalability Over Large Graphs

As scalability can be a huge challenge for data-driven approaches to the hard combinatorial problems, we also perform extensive time-complexity analysis on large graphs. It is important to show that a method that is much faster than the state-of-the-art methods, does not compromise on results as the size of the input increases. Hence, we train our GCN model on the small graphs and then test them on larger graphs (almost 10 times the size of training graphs) and observe the performance.

In this experiment setup, we combine all the available datasets and sort the graphs by their size. We split this sorted mixture such that the smallest 70% of the graphs are in the train set and the largest 15% are in the test set. The graph sizes in the training set range from $6-110$. It takes the greedy algorithm less than 10 secs to find the solution for a graph in this size range. The graphs in the test set range from $500-1194$ in size. Using the same rest of the experimental setup as in Section VI-D, we evaluate the proposed approach on the accuracy and time complexity metrics. The sizes of ZFS $\zeta_{gr}(G)$, and $\zeta_{gcn}(G)$, and the time taken for ZFS computation of each large graph in the test set are plotted in Fig. 9.

Interestingly, our proposed approach turns out to give a solution that is nearly as same in size as the greedy solution on these large graphs while taking only a fraction of time. The average deviation for these large test graphs is 0.745% with a maximum of 3.4% (19 vertices less than the greedy solution) and a minimum of -9.2% (29 vertices more than the greedy



(a) (b)

Fig. 10. Comparison between random-greedy and the proposed GCN-based solution. Plot (a) shows the deviation from the greedy solution with varying randomness threshold τ, and plot (b) presents the respective time taken to compute the solution on the whole dataset.

solution) deviation. The average size of the greedy solution is 449.3 vertices in the test set. This indicates that on average, there are about 3.5 more vertices in the $Z(G)$ obtained by the GCN than in the greedy solution. However, the GCN takes a total time of about 2.5 days (primarily the time for the redundancy check) while the greedy algorithm takes about 48 days to compute the $Z(G)$ for the 1832 graphs in the test set. This essentially means that *the GCN architecture is less than 1% away from the solution while being about 19 times faster on large graphs*.

We expect the time difference between GCN and greedy algorithm to grow exponentially for larger graphs. However, validating this hypothesis for significantly larger graphs becomes impractical, primarily due to the fact that the greedy algorithm's computational demands become prohibitively high on graphs with more than 2500 vertices. As shown above, GCN algorithm is computationally much faster than the greedy algorithm on large graphs while preserving the solution quality. Even though random selection and greedy heuristics can reduce time, GCN is still significantly faster than the greedy algorithm, especially for larger graphs. To substantiate this, we conduct an empirical study focusing on computational efficiency and solution quality. Our experimentation involved varying the randomness threshold τ of the initially selected vertices from 35% to 75%. To get a range of sizes of the graphs, we select the REDDIT-BINARY graphs and a part of the Large ER graphs dataset to obtain graphs with the number of vertices ranging between $6 - 1194$. We compare the quality of the solution using Dev(ζ_{rdm}) and Dev(ζ_{gcn}). Fig. 10 compares the random-greedy with the proposed GCN-based approach.

We observe that the random-greedy algorithm consumes significantly more time than our GCN-based approach, even when allowing for up to 75% random selection. This underscores the computational advantages of our method, especially for larger graphs. Fig. 10(a) demonstrates that increasing the randomness threshold τ slightly deteriorates the solution quality for both approaches. This trade-off is expected, but our approach consistently maintains a competitive solution quality. Additionally, the time taken by the random selection and the greedy heuristics decreases with the increase of threshold τ as it takes fewer iterations of the greedy algorithm to complete the solution. Conversely, our GCN-based approach incurs a slight increase in time as τ rises due to the overhead cost of redundancy checks, which are performed in a greedy fashion. In sum, our empirical findings collectively support the argument that even in cases where random selection and greedy heuristics are employed, *the random-greedy approach lags significantly behind the computational efficiency of our GCN-based method, particularly as graph size increases.*

IX. Conclusion

Minimum ZFS is a hard combinatorial optimization problem with several applications across various domains. We presented a novel graph convolutional network to compute a small-sized ZFS. The proposed solution utilizes data-driven and algorithmic insights to compute ZFS in large graphs effectively. Through extensive experiments, we showed that the proposed approach is computationally efficient, scalable to much larger graphs, generalizable to different graph families, and able to learn from sub-optimal datasets. Thus, our approach is not inhibited by the requirement to have optimal datasets consisting of large enough instances to train the graph learning models for combinatorial optimization problems. We also contributed towards the future data-driven algorithms for the minimum ZFS problem through several synthetic and real-world benchmark datasets using greedy solutions as the ground truth. In addition, a small graphs dataset containing hard instances is also annotated with optimal solutions. In the future, we aim to extend our approach to solving other combinatorial optimization problems, including minimum dominating sets in graphs.

References

[1] N. Barnier and P. Brisset, "Graph coloring for air traffic flow management," *Ann. Operations Res.*, vol. 130, no. 1, pp. 163–178, 2004.
[2] Y. Peng, B. Choi, B. He, S. Zhou, R. Xu, and X. Yu, "VColor: A practical vertex-cut based approach for coloring large graphs," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 97–108.
[3] B. Balasundaram and S. Butenko, "Graph domination, coloring and cliques in telecommunications," in *Handbook of Optimization in Telecommunications*. Berlin, Germany: Springer, 2006, pp. 865–890.
[4] F. Moradi, T. Olovsson, and P. Tsigas, "A local seed selection algorithm for overlapping community detection," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining*, 2014, pp. 1–8.
[5] N. Armenatzoglou, H. Pham, V. Ntranos, D. Papadias, and C. Shahabi, "Real-time multi-criteria social graph partitioning: A game theoretic approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1617–1628.
[6] P. R. Östergård, "A fast algorithm for the maximum clique problem," *Discrete Appl. Math.*, vol. 120, no. 1–3, pp. 197–207, 2002.
[7] D. Burgarth, D. D'Alessandro, L. Hogben, S. Severini, and M. Young, "Zero forcing, linear and quantum controllability for systems evolving on networks," *IEEE Trans. Autom. Control*, vol. 58, no. 9, pp. 2349–2354, Sep. 2013.
[8] B. Brimkov, C. C. Fast, and I. V. Hicks, "Computational approaches for zero forcing and related problems," *Eur. J. Oper. Res.*, vol. 273, no. 3, pp. 889–903, 2019.
[9] P. A. Dreyer Jr. and F. S. Roberts, "Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion," *Discrete Appl. Math.*, vol. 157, no. 7, pp. 1615–1627, 2009.
[10] N. Monshizadeh, S. Zhang, and M. K. Camlibel, "Zero forcing sets and controllability of dynamical systems defined on graphs," *IEEE Trans. Autom. Control*, vol. 59, no. 9, pp. 2562–2567, Sep. 2014.
[11] S. S. Mousavi, M. Haeri, and M. Mesbahi, "On the structural and strong structural controllability of undirected networks," *IEEE Trans. Autom. Control*, vol. 63, no. 7, pp. 2234–2241, Jul. 2018.

[12] A. Aazami, "Hardness results and approximation algorithms for some problems on graphs," Ph.D. dissertation, University of Waterloo, Waterloo, ON, Canada, 2008.
[13] W. Abbas, M. Shabbir, Y. Yazıcıoğlu, and X. Koutsoukos, "Leader selection for strong structural controllability in networks using zero forcing sets," in *Proc. Amer. Control Conf.*, 2022, pp. 1444–1449.
[14] F. Gama, E. Tolstaya, and A. Ribeiro, "Graph neural networks for decentralized controllers," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, 2021, pp. 5260–5264.
[15] M. Boffa, Z. B. Houidi, J. Krolikowski, and D. Rossi, "Neural combinatorial optimization beyond the TSP: Existing architectures under-represent graph structure," in *Proc. AAAI Workshop Graphs More Complex Structures Learn. Reasoning*, 2022.
[16] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," *J. Mach. Learn. Res.*, vol. 24, no. 130, pp. 1–61, 2023.
[17] M. Böther, O. Kißig, M. Taraz, S. Cohen, K. Seidel, and T. Friedrich, "What's wrong with deep learning in tree search for combinatorial optimization," in *Proc. Int. Conf. Learn. Representations*, 2022, pp. 1–25.
[18] K. Xu, M. Zhang, J. Li, S. S. Du, K. Kawarabayashi, and S. Jegelka, "How neural networks extrapolate: From feedforward to graph neural networks," in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–52.
[19] C. K. Joshi, T. Laurent, and X. Bresson, "An efficient graph convolutional network technique for the travelling salesman problem," 2019, *arXiv:1906.01227*.
[20] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2016, *arXiv:1611.09940*.
[21] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Velickovic, "Combinatorial optimization and reasoning with graph neural networks," *J. Mach. Learn. Res.*, vol. 24, no. 130, pp. 1–61, 2023.
[22] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–25.
[23] Q. Wang and C. Tang, "Deep reinforcement learning for transportation network combinatorial optimization: A survey," *Knowl.-Based Syst.*, vol. 233, 2021, Art. no. 107526.
[24] AIM Minimum Rank Special Graphs Work Group, "Zero forcing sets and the minimum rank of graphs," *Linear Algebra Appl.*, vol. 428, no. 7, pp. 1628–1648, 2008.
[25] M. Trefois and J.-C. Delvenne, "Zero forcing number, constrained matchings and strong structural controllability," *Linear Algebra Appl.*, vol. 484, pp. 199–218, 2015.
[26] A. Chapman and M. Mesbahi, "On strong structural controllability of networked systems: A constrained matching approach," in *Proc. Amer. Control Conf.*, 2013, pp. 6126–6131.
[27] S. Fallat, K. Meagher, and B. Yang, "On the complexity of the positive semidefinite zero forcing number," *Linear Algebra Appl.*, vol. 491, pp. 101–122, 2016.
[28] I. V. Hicks et al., "Computational and theoretical challenges for computing the minimum rank of a graph," *INFORMS J. Comput.*, vol. 34, pp. 2868–2872, 2022.
[29] D. Ferrero et al., "Rigid linkages and partial zero forcing," *Elec. J. Combinatorics*, pp. P2–43, 2019.
[30] F. H. Kenter and J. C.-H. Lin, "On the error of a priori sampling: Zero forcing sets and propagation time," *Linear Algebra Appl.*, vol. 576, pp. 124–141, 2019.
[31] J. C.-H. Lin, P. Oblak, and H. Šmigoc, "The strong spectral property for graphs," *Linear Algebra Appl.*, vol. 598, pp. 68–91, 2020.
[32] D. Burgarth and V. Giovannetti, "Full control by locally induced relaxation," *Phys. Rev. Lett.*, vol. 99, no. 10, 2007, Art. no. 100501.
[33] S. Butler et al., "Minimum rank library. Sage programs for calculating bounds on the minimum rank of a graph, and for computing zero forcing parameters," 2014, Accessed: Mar. 24, 2022. [Online]. Available: https://github.com/jasongrout/minimum_rank
[34] B. Brimkov, D. Mikesell, and I. V. Hicks, "Improved computational approaches and heuristics for zero forcing," *INFORMS J. Comput.*, vol. 33, pp. 1259–1684, 2021.
[35] A. Agra, J. O. Cerdeira, and C. Requejo, "A computational comparison of compact MILP formulations for the zero forcing number," *Discrete Appl. Math.*, vol. 269, pp. 169–183, 2019.
[36] A. Weber, G. Reissig, and F. Svaricek, "A linear time algorithm to verify strong structural controllability," in *Proc. IEEE 53rd Conf. Decis. Control*, 2014, pp. 5574–5580.
[37] Y. Jiang, Y. Rong, H. Cheng, X. Huang, K. Zhao, and J. Huang, "Query driven-graph neural networks for community search: From non-attributed, attributed, to interactive attributed," *Proc. VLDB Endowment*, vol. 15, no. 6, pp. 1243–1255, 2022.
[38] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, vol. 2, 2015, pp. 2692–2700.
[39] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 537–546.
[40] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2015, pp. 1365–1374.
[41] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.
[42] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann, "TUDataset: A collection of benchmark datasets for learning with graphs," 2020, *arXiv:2007.08663*.
[43] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2016.
[44] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

Obaid Ullah Ahmad received the B.S. degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 2019, and the M.S. degree in computer science from Information Technology University, Lahore, in 2021. He is currently working toward the Ph.D. degree in electrical engineering from the University of Texas at Dallas, Richardson, TX, USA. He is currently a Research Assistant with the University of Texas at Dallas' Control, Intelligence, Resilience in Networks and Systems Lab, Richardson. His current research interests include control-based approaches, for graph machine learning, network optimization, and multi-robot systems.

Mudassir Shabbir received the Ph.D. degree from the Division of Computer Science, Rutgers University, New Brunswick, NJ, USA, in 2014. He is currently an Associate Professor with the Department of Computer Science, Information Technology University, Lahore, Pakistan, and a Research Assistant Professor with Vanderbilt University, Nashville, TN, USA. He was with the Lahore University of Management Sciences, Pakistan; Los Alamos National Labs, NM, Bloomberg L.P. New York, NY, USA; and with Rutgers University. He was Rutgers Honors Fellow for 2011 to 2012. His main research interests include algorithmic and discrete geometry, and has developed new methods for the characterization and computation of succinct representations of large data sets with applications in non-parametric statistical analysis. He also works on graph machine learning and resilient network systems.

Waseem Abbas (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2010 and 2013, respectively. He is currently an Assistant Professor with the System Engineering Department, University of Texas at Dallas, Richardson, TX, USA. He was a Research Assistant Professor with the Vanderbilt University, Nashville, TN, USA. He was a Fulbright Scholar from 2009 to 2013. His research interests include control of networked systems, resilience and robustness in networks, distributed optimization, and graph-theoretic methods in complex networks.

Xenofon Koutsoukos (Fellow, IEEE) received the Ph.D. degree in electrical engineering from the University of Notre Dame, Notre Dame, IN, USA, in 2000. He is currently the Thomas R. Walters Professor and Chair of the Department of Computer Science, School of Engineering, Vanderbilt University, Nashville, TN, USA. He is also a Senior Research Scientist with the Institute for Software Integrated Systems (ISIS) and holds a secondary appointment with the Department of Electrical and Computer Engineering. He was a Member of Research Staff with the Xerox Palo Alto Research Center (PARC) during 2000–2002. He has coauthored more than 350 journal and conference papers, and he is co-inventor of four U.S. patents. His research interests include cyber-physical systems with emphasis on learning-enabled systems, security and resilience, diagnosis and fault tolerance, distributed algorithms, formal methods, and adaptive resource management. Dr. Koutsoukos was the recipient of the NSF Career Award in 2004, Excellence in Teaching Award in 2009 from the Vanderbilt University School of Engineering, and the 2011 NASA Aeronautics Research Mission Directorate (ARMD) Associate Administrator (AA) Award in Technology and Innovation.