Graph Problems with Graph Neural Networks

August Femtehjell

Department of Mathematics

(Dated: June 2024)

This paper utilized Graph Convolutional Networks (GCNs) in order to solve combinatorial graph problems. This is done physics-informing them, exploiting the deep connection between Ising glass models and Quadratic Unconstrained Binary Optimization. This allows the models performance to be measured by a differentiable function, which doubles as the loss function of the network. This framework is able to outperform a simple approximate algorithm for the Minimum Vertex Covering (MVC) problem, however struggles with the more complex Traveling Salesman Problem (TSP).

CONTENTS

I. INTRODUCTION

Graphs serve as one of the most fundamental and natural ways to express relational data. In their simplest form, they are made up of a set of vertices V, where some vertices are connected to each other from edges in a set E. This forms a graph $G = (V, E)$. With this simple framework, one is able to describe a large number of phenomena, from social circles and road networks to the complex interactions arising in molecules.

The loose structure of a graph poses a problem in connection with typical machine learning methods, as they typically rely on the data being organised in regular arrays. Within a graph, two vertices may be connected to a different number of vertices, meaning the data structure is non-euclidean. A special class of machine learning methods has therefore been devised, dubbed Graph Neural Networks (GNNs) (Wu *et al.*, 2021), in order to capture the inherent relations between the data in graphs.

A major factor in the predictive power of machine learning models, is the size of the dataset which the model was trained on. In a lot of domains, the available data is often limited, or expensive to gather. Consider for instance experimental data from a wave pool. Although the measured data might be too limited to fully capture the behaviour, the governing mechanics are well known. In order to capitalise on this, Physics-Informed Neural Networks (PINNs) have been developed (Raissi *et al.*, 2019), which incorporate the underlying Partial Differential Equations (PDEs). They do this through differentiating the network under the same conditions as in the PDE, ensuring that the trained model not only predicts correctly within the training set, but also acts as a surrogate for the true solution elsewhere.

Combinatorial optimization problems concern themselves with choosing the optimal selection from a finite set of feasible solutions. A famous example of this is the Traveling Salesman Problem (TSP), where the goal is to choose a route between a set of cities, such that the total traveled distance is minimal. The complexity of this problem stems from the fact that with n cities, there are $n!$ different ways to visit the cities. Because of this, no efficient algorithm has been found which finds the optimal route for a given graph, meaning the available training data is rather limited. This is not an issue exclusive to TSP, and is present in a whole range of problems in a class called NP, where there exists a polynomial time algorithm for verifying the correctness of a solution, but not one for finding the solution.

In order to overcome the lack of data for these problems, the constraints of a problem are encoded in a Hamiltonian, such that a trial solution can be evaluated without apriori knowledge of the optimal solution. Schuetz *et al.* (2022) utilized this method for solving the Maximum cut (MaxCut) and Maximum Independent Set (MIS), where each node is assigned a soft probability, providing an approximate solution. This Hamiltonian then serves as a differentiable cost function, in effect physics-informing the GNN.

In this report, I will briefly explain the basics of Graph Theory and GNNs, before applying this framework to the Minimal Vertex Cover (MVC) and Traveling Salesman problems.

II. THEORY

A. A brief introduction to Graph Theory

There are a number of different types of graphs, with the simplest being dubbed the *simple* graph. It is simple in the sense that it is undirected, and at most contains a single edge between two nodes. Additionally, there exists no edge from a node to itself. Here, a graph is meant as a simple graph, unless otherwise specified.

Associated with a graph is the adjacency matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, defined such that $a_{uv} = 1$ if there exists an edge between nodes u and v, and $a_{uv} = 0$ otherwise. Here, $n = |V|$, i.e. the number of nodes in the graph.

With some graphs we additionally define a weight function $w : E \to \mathbb{R}$, assigning a weight to each of the edges. In a graph of cities, this function might then represent the length of a road connecting two cities.

An important measure when working with the complexity of problems, comes from asymptotic analysis. For a function g, we define $\mathcal{O}(g(n))$ as the set of all functions growing slower than (or equal) to g. Formally, we write

$$\mathcal{O}(g(n)) = \{f(n) : \exists C > 0 \text{ and } n_0 \in \mathbb{N} \text{ s.t. } |f(n)| \leq C|g(n)| \ \forall n \geq n_0\}. \quad (1)$$

Thus, for instance we have $n \in \mathcal{O}(n^2)$. Although this notation is formally correct, it is traditionally written $n = \mathcal{O}(n^2)$ (Aigner, 2023, p. 100). Note that this is not a precise measure of growth, but rather an upper bound.

Returning to the graph of cities, a common problem one might encounter is to determine the shortest route connecting two cities. The number of steps required to solve this problem is bounded by $\mathcal{O}(|E|+|V|\log|V|)$ with Dijkstra's Algorithm (Fredman and Tarjan, 1984), growing linearithmic with the size of the graph. In problem complexity, the 'hardness' of a problem is typically defined by whether or not there exists an algorithm which solves the problem in polynomial time. A problem of this difficulty is said to be in class P. As $n \log n = \mathcal{O}(n^2)$, the shortest path problem is 'easily' solved.

An interesting class of problems are those in NP, meaning *non-deterministic polynomial time*. These problems are defined by there being a polynomial time algorithm for verifying a solution, however not necessarily one to find the solution. Both Sudoku (of arbitrary board size) and the TSP fall into this category, among many others.

B. Graph Neural Networks

Graph Neural Networks (GNNs) are a class of deep learning methods designed to operate on graphs. They do this in various ways in order to utilize the inherent graph structure of the data. One such method is the Graph Convolutional Network (GCN) (Kipf and Welling, 2017), which can be viewed as a generalization of Convolutional Neural Networks (CNNs).

CNNs are designed to operate on euclidean data, which typically consists of 1-D or 2-D grids (Goodfellow *et al.*, 2016, p. 326–339). A convolutional layer is typically made up of three different stages. Firstly, there's the convolutional stage where an affine transformation is applied, related to how one would convolve two functions. Given an input matrix I and a matrix called the kernel, the resulting convolution S is defined by

$$S_{i,j} = (K * I)_{i,j} = \sum_m \sum_n I_{i-m,j-n} K_{m,n}. \quad (2)$$

Typically, the dimension of the kernel is much smaller than that of the input. In a machine learning context, the kernel typically contains trainable weights, allowing for expressivity in how an element is affected by nearby values.

Next, non-linearity is introduced through an activation function. A typical choice for CNNs is the rectified linear unit (ReLU), defined as $\text{ReLU}(x) = \max\{x, 0\}$. Finally, there is a pooling layer, where each element is replaced by the result of some operation applied to the neighbouring elements, illustrated in Figure 1. This could for instance be max pooling, where the result is the maximal value of the neighbouring elements, or simply the sum.

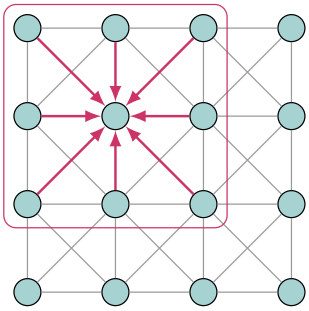

FIG. 1: An illustration of a typical convolutional layer. Communication happens between elements within a fixed area.

These convolutions are dependent on the structure of the data being both rigid and known. Arranging the data as a graph, as in Figure 1, kernel-level communication can be described using a weighted directed graph. In this way, one gets a sense of how convolutions can be extended to arbitrary graphs, where communication happens across the 1^{st}-order neighbourhood of a node.

GCNs update node features through the propagation rule

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right), \quad (3)$$

where $\tilde{A} = A + I_n$ denotes the adjacency matrix of the graph with added self-edges, $\tilde{D}_{i,i} = \sum_j A_{i,j}$ is the matrix

with diagonal elements equal to the outdegree of each node, $H^{(l)}$ and $W^{(l)}$ are respectively the activated values and the trainable weights for the l^{th} layer, with $H^{(0)}$ equal the input graph. $\sigma : \mathbb{R} \to \mathbb{R}$ denotes the chosen activation function for the layer, applied elementwise.

Note that $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is a way to normalize the values in $\tilde{A}$, derived from an approximation of spectral graph convolutions. See Kipf and Welling (2017) and Hammond *et al.* (2009) for a more detailed explanation of this approximation.

An important property of adjacency matrices is that $(A^k)_{i,j}$ is the number of paths from node i to node j of length k (Aigner, 2023, p. 125), meaning that repeated applications of (3) results in communication over the k^{th}-order neighbourhood around a node. One such application around a node is visualized in Figure 2.

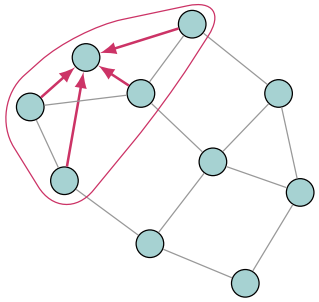

FIG. 2: An application of a GCN layer, with communication across the 1^{st}-order neighbourhood.

In practice, the affine transformation $H^{(l)}W^{(l)}$ in (3) can be replaced with a more complex function, such as a multilayer perceptron (MLP). This might grant the model more expressivity. Note that this operates elementwise for each node feature, with shared parameters.

One problem with GCNs is that they do not function on complete graphs. This stems from the fact that with a fully connected graph, the adjacency matrix contains a 1 everywhere but the diagonal. Then, $\tilde{A} = A + I_n$ is a matrix of all ones, resulting in $\tilde{D}_{i,i} = n$. Multiplying a matrix by $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ thus causes all node features to result in the same vector, a concept known as oversmoothing (Rusch *et al.*, 2023). This can also occur with sparser graphs if too many message passing layers are added.

C. Ising formulations

The Ising model was originally created in order to model the properties of ferromagnets, although it has since found a number of different applications. The model considers in the simplest case a string of magnetic moments, each assigned either *up* or *down* spin (Singh, 2020). An Ising model thus consists of N spins $s_i = \pm 1$, together with a Hamiltonian

$$H(s_1, \ldots, s_N) = -\sum_{i<j} J_{ij} s_i s_j - \sum_{i=1}^{N} h_i s_i, \tag{4}$$

for $J_{ij}, h_i \in \mathbb{R}$ (Lucas, 2014).

Through the function $f(x) = \frac{1}{2}(x+1)$, the spins can be translated to binary decision variables, such that $x_i = f(s_i) \in \{0, 1\}$. The Ising model is thus closely related to Quadratic Unconstrained Binary Optimization (QUBO) problems, defined as the programming problem

$$\begin{aligned} &\text{maximize} \quad x^T Q x + c^T x \\ &\text{subject to} \quad x \in \{0,1\}^n, \end{aligned} \tag{5}$$

for a given real matrix Q and vector c (Punnen, 2022, p. 1–7). Often, Q is upper triangular, meaning we can rewrite the objective function in (5) as

$$\sum_{i \le j} Q_{ij} x_i x_j + \sum_{i=1}^{n} c_i x_i. \tag{6}$$

Additionally, maximizing an objective function is equivalent with minimizing the negative of the objective function, meaning we can rewrite (5) as

$$\begin{aligned} &\text{minimize} \quad -\sum_{i<j} Q_{ij} x_i x_j - \sum_{i=1}^{n} c_i x_i \\ &\text{subject to} \quad x \in \{0,1\}^n, \end{aligned} \tag{7}$$

which is immediately recognizable as the Hamiltonian in (4).

With this framework, one can encode a number of different optimization and related decision problems. Lucas (2014) formulates Hamiltonians for a number of different NP problems, including all 21 of Karp's NP-Complete problems (Karp, 2010), with the intended aim of utilizing quantum adiabatic optimization.

However, relaxing the constraint that $x_i \in \{0, 1\}$ to $x_i \in [0, 1]$, Schuetz *et al.* (2022) notes that these Hamiltonians can serve as differentiable loss functions for a neural network. With this, one can reduce the need for labeled data in the training process of the neural network, in effect *physics-informing* them. This is especially helpful when working with NP problems, as labeled data is often in low supply, or missing entirely, due to the complexity of the problems.

1. Minimal Vertex Covering

A vertex cover is a coloring of the nodes in a graph, such that each edge is connected to at least one colored node. The aim of the Minimal Vertex Covering (MVC) problem is to find a coloring such that the number of colored nodes in minimal.

Lucas (2014) formulates this as an Ising problem using a bitstring $x \in \mathbb{R}^n$, where $x_v = 1$ if node v is colored, and 0 otherwise. The constraint that each edge must be colored is then enforced through

$$H_\alpha = \sum_{uv \in E} (1 - x_u)(1 - x_v), \tag{8}$$

where we have $H_\alpha = 0$ if and only if each edge is connected to at least one colored node. The 'minimal' aspect is enforced through appending the term

$$H_\beta = \sum_{u \in V} x_u. \tag{9}$$

The total Hamiltonian is then $H = \alpha \cdot H_\alpha + \beta \cdot H_\beta$, where choosing $\alpha > \beta$ ensures that the global minima is optimal for the problem.

In order to improve computational performance, we typically want to avoid using explicit for-loops. This stems from the fact that we would rather want to use packages designed for numerical linear algebra, which have been optimized for vector-matrix operations over the last decades.

Letting $\bar{x}_u = 1 - x_u$, we utilize the outer product to get

$$\bar{X} = \bar{x}\bar{x}^T = \begin{bmatrix} \bar{x}_1^2 & \bar{x}_1\bar{x}_2 & \cdots & \bar{x}_1\bar{x}_n \\ \bar{x}_2\bar{x}_1 & \bar{x}_2^2 & & \bar{x}_2\bar{x}_n \\ \vdots & & \ddots & \vdots \\ \bar{x}_n\bar{x}_1 & \bar{x}_n\bar{x}_2 & \cdots & \bar{x}_n^2 \end{bmatrix}, \tag{10}$$

such that $\bar{X}_{uv} = (1 - x_u)(1 - x_v)$. With a simple graph, i.e. a graph which is undirected and contains no parallel edges or self-loops, and A as the adjacency matrix we then have that

$$\mathbb{1}_n \left(A \odot \bar{X}\right) \mathbb{1}_n = 2 \sum_{uv \in E} (1 - x_u)(1 - x_v), \tag{11}$$

as both $\hat{X}_{uv}$ and $\hat{X}_{vu}$ represent the constraint for a single edge. Here, $\mathbb{1}_n = [1,\ 1,\ \ldots,\ 1] \in \mathbb{R}^n$, and $\odot$ denotes the elementwise Hadamard product.

Thus,

$$H = \mathbb{1}_n^T \left(A \odot \bar{X}\right) \mathbb{1}_n + \mathbb{1}_n^T x \tag{12}$$

is equivalent with the original Hamiltonian with $\alpha = 2$ and $\beta = 1$, satisfying the requirement that $\alpha > \beta$.

2. Traveling Salesman Problem

In a graph $G = (V, E)$, a Hamiltonian cycle is a path which starts and ends in the same node, passing through all other nodes in the graph. Lucas (2014) encodes these constraints in a bitstring $x_{v,i}$ of size n^2, where $n = |V|$. $x_{v,i} = 1$ denotes that the v^{th} node should be placed in the i^{th} place in the cycle. Thus, a viable cycle would be one which includes each vertex exactly once, with exactly one vertex at each place in the cycle.

A bitstring is thus considered feasible if

$$\sum_{v=1}^{n} \left(1 - \sum_{i=1}^{n} x_{v,i}\right)^2 + \sum_{i=1}^{n} \left(1 - \sum_{v=1}^{n} x_{v,i}\right)^2 = 0. \tag{13}$$

This would be enough if we assume that the graph is *complete*, i.e. if there exists an edge between each vertex pair. However, this might not be the case for a given graph. Thus, the Hamiltonian associated with guaranteeing a Hamiltonian cycle[1] is

$$\begin{aligned} H_\alpha = &\sum_{v=1}^{n} \left(1 - \sum_{i=1}^{n} x_{v,i}\right)^2 + \sum_{i=1}^{n} \left(1 - \sum_{v=1}^{n} x_{v,i}\right)^2 \\ &+ \sum_{uv \notin E} \sum_{i=1}^{n} x_{u,i} x_{v,i+1}, \end{aligned} \tag{14}$$

with the convention that $x_{v,n+1} = x_{v,0}$.

Extending this to also formulate TSP is now simple, by adding a term H_β defined as

$$H_\beta = \sum_{uv \in E} W_{uv} \sum_{i=1}^{n} x_{u,i} x_{v,i+1}, \tag{15}$$

with W_{uv} being the weight associated with the edge uv. The total Hamiltonian for TSP is then

$$H = \alpha \cdot H_\alpha + \beta \cdot H_\beta, \tag{16}$$

with coefficients $\alpha, \beta > 0$. As it should always be favourable to add a heavy edge in order to ensure that the solution is a Hamiltonian cycle, we require that $\beta \max_{uv \in E} W_{uv} < \alpha$.

With graphs, it is typically easier to work with the adjacency matrix. As the bitstring $x_{v,i}$ only gives the placements of the nodes in a cycle, it is not immediately clear how one is supposed to retrieve the predicted adjacency matrix. However, we were already close to this when formulating the Hamiltonian, as there exists an edge between u and v if $x_{u,i} x_{v,i+1} > 0$ for some i. We therefore define the predicted adjacency matrix $\hat{A} = (\hat{a}_{uv}) \in \mathbb{R}^{n \times n}$ such that

$$\hat{a}_{uv} = \sum_{i=1}^{n} x_{u,i} x_{v,i+1}. \tag{17}$$

Again, (16) should ideally be described in terms of matrix operations. With the bitstring $x_{v,i}$ we associate

[1] Both are named after Sir William Rowan Hamilton.

the matrix X defined as

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & & x_{2,n} \\ \vdots & & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix}. \tag{18}$$

Then, (17) corresponds with the matrix multiplication

$$\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & & x_{2,n} \\ \vdots & & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix} \begin{bmatrix} x_{1,2} & x_{2,2} & \dots & x_{n,2} \\ x_{1,3} & x_{2,3} & & x_{n,3} \\ \vdots & & \ddots & \vdots \\ x_{1,n} & x_{2,n} & \dots & x_{n,n} \\ x_{1,1} & x_{2,1} & & x_{n,1} \end{bmatrix}. \tag{19}$$

The second matrix corresponds with shifting the rows of the transpose of X, giving us

$$\hat{A} = XPX^T, \tag{20}$$

where $P \in \mathbb{R}^{n\times n}$ is the permutation matrix

$$P = \begin{bmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ 1 & & & \end{bmatrix}. \tag{21}$$

With $W \in \mathbb{R}^{n\times n}$ being the matrix of weights W_{uv}, with zeroes where there is not an edge, we can express the edge related terms of (16) as

$$\begin{aligned} \sum_{uv\notin E} \sum_{i=1}^{n} x_{u,i}x_{v,i+1} &= \mathbb{1}_n^T \left(T \odot \hat{A}\right) \mathbb{1}_n, \\ \sum_{uv\in E} W_{uv} \sum_{i=1}^{n} x_{u,i}x_{v,i+1} &= \mathbb{1}_n^T \left(W \odot \hat{A}\right) \mathbb{1}_n, \end{aligned} \tag{22}$$

where $T = (t_{uv}) \in \mathbb{R}^{n\times n}$ is the matrix defined through

$$t_{uv} = \begin{cases} 1 & \text{if } W_{uv} = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{23}$$

The first terms of (14) are

$$\begin{aligned} \sum_{v=1}^{n} \left(1 - \sum_{i=1}^{n} x_{v,i}\right)^2 &= \left(\mathbb{1}_n - \mathbb{1}_n^T X\right)^2, \\ \sum_{i=1}^{n} \left(1 - \sum_{v=1}^{n} x_{v,i}\right)^2 &= \left(\mathbb{1}_n - X\mathbb{1}_n\right)^2, \end{aligned} \tag{24}$$

giving us all the terms needed to efficiently implement (16). The Hamiltonian for TSP is thus defined through

$$\begin{aligned} H_\alpha &= \left(\mathbb{1}_n - \mathbb{1}_n^T X\right)^2 + \left(\mathbb{1}_n - X\mathbb{1}_n\right)^2 \\ &\quad + \mathbb{1}_n^T \left(T \odot \hat{A}\right) \mathbb{1}_n, \\ H_\beta &= \mathbb{1}_n^T \left(W \odot \hat{A}\right) \mathbb{1}_n. \end{aligned} \tag{25}$$

III. METHOD

A. The JAX ecosystem

The network is implemented through the `JAX` ecosystem (Bradbury *et al.*, 2018), in order to improve the efficiency of the computations in Python. It does this through implementing just-in-time compilation (jit), automatic differentiation and utilizing XLA (Accelerated Linear Algebra) as its backend.

The graphs are generated using `NetworkX` (Hagberg *et al.*, 2008), and translated to `Jraph` (Godwin *et al.*, 2020) which allows for sparse graph computations in `JAX`. `Jraph` serves as an extension of `FLAX` (Heek *et al.*, 2023), which is a neural network library developed by Google for `JAX`, implementing a number of graph neural network utilities. The optimization is performed using the `optax` library (DeepMind *et al.*, 2020), with Adam (Kingma and Ba, 2017) as the optimizer.

B. Model architecture

The neural network consists of k initial graph convolutions, each incorporating their own MLP with q layers. The leaky ReLU function is used as the hidden activation function, defined as $\max\{0.01x, x\}$. The weights of the MLPs are initialized through He initialization (He *et al.*, 2015), where the weights are drawn from the normal distribution

$$w_l \sim \mathcal{N}(0, 2/n_l), \tag{26}$$

where n_l is the number of inputs in the l^{th} layer.

The cost functions are as described the problem specific Hamiltonians, where the outputs from the neural network are the predicted bitstring x. The softmax or sigmoid function is used as the output activation in order to ensure valid probabilities. The probabilities are then finally rounded to the nearest of 0 and 1 in the evaluation stage in order to retrieve the approximate solution.

IV. DISCUSSION AND RESULTS

A. Minimum Vertex Cover

The network consists of two graph convolutional layers, each consisting of an MLP with 1 layer of 100 hidden nodes. This the output is activated through the sigmoid function, through a final graph convolution.

Firstly, the model is trained on a singular graph. This graph is randomly generated through the `networkx` package, consisting of 100 nodes. Each edge has a 5% probability of being included in the graph. After 653 epochs, the model converges, taking approximately 1 second. Note that the models are run in a Jupyter Notebook on an M1

mac, and the runtimes are therefore not directly comparable with other methods.

The vertex cover is shown in Figure 3, having a size of 61 coloured nodes, satisfying all constraints. Additionally the number of unnecessary nodes are computed, being the number of colored nodes only connected to other colored nodes.

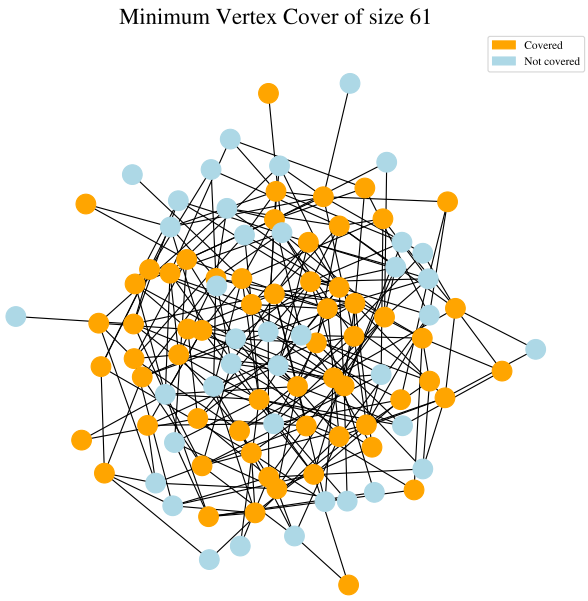

FIG. 3: Model trained on a single random graph. Orange nodes are covered, blue are not.

On such a small sample, the model is likely to be sensitive to the initial conditions of the model. Therefore, the model is initialised on 100 random seeds, each trained on the same initial graph. The results are presented in Table I, showing that the model is consistently predicting a cover of around 61.

TABLE I: Model sensitivity to initial seed.

# of occurrences	size	edge violations	unnecessary colorings
69	61	0	0
19	60	0	0
9	62	0	0
1	59	0	0
1	62	0	1
1	99	0	39

The predicted cover sizes are compared against the local-ratio algorithm (Bar-Yehuda and Even, 1985), guaranteeing an approximate solution which is at most twice as large as the optimal covering. For this initial graph, the approximate cover consists of 80 nodes, meaning our model is able to outperform it.

Finally, the model is trained on two sets of 100 graphs, with the first being random with probability 5%, and the second being 3-regular. The models then predicts a cover for two sets of unseen graphs, being another set of random and regular graphs. In Figure 4 we see that the model is able to consistently predict valid solutions, beating the approximate solution in all cases. However, it is not able to generalise well for unseen graphs.

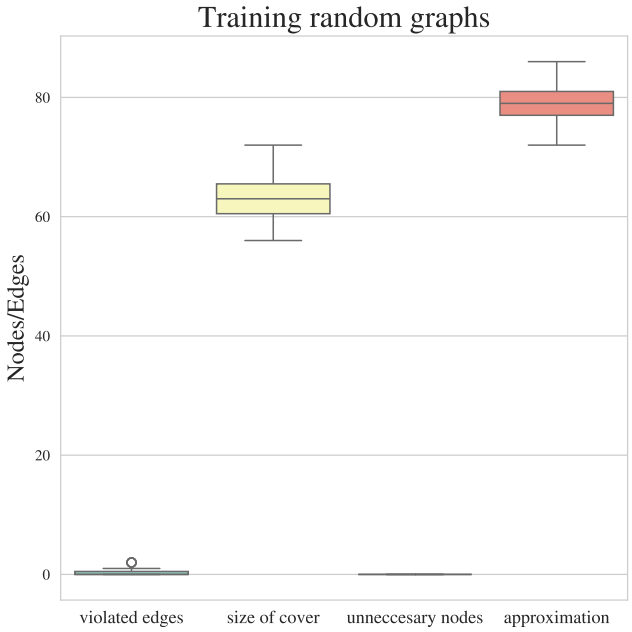

FIG. 4: MVC results from training on 100 random graphs.

In Figure 5 and Figure 6, this model is evaluated against respectively random and regular graphs. The model is rarely able to predict valid solutions, and those which are valid are worse than the approximations. Notably, the model seems to perform better on the 3-regular graphs than on the random ones.

Training the model on 3-regular graphs instead, as in Figure 7, the model is again able to outperform the approximations. It seems to perform slightly worse unseen 3-regular graphs in Figure 8, and a lot worse on the unseen random graphs in Figure 9.

Due to memory constraints, the size of the training set was rather limited, perhaps explaining the poor generalisation. However, given the fact that the generated solutions are purely attained through training the models without prior knowledge, the results from the training sets are perhaps a valid measure of the models performance.

B. Hamiltonian cycles

Although the objectives of finding a Hamiltonian cycle in a graph and determining the route of a traveling

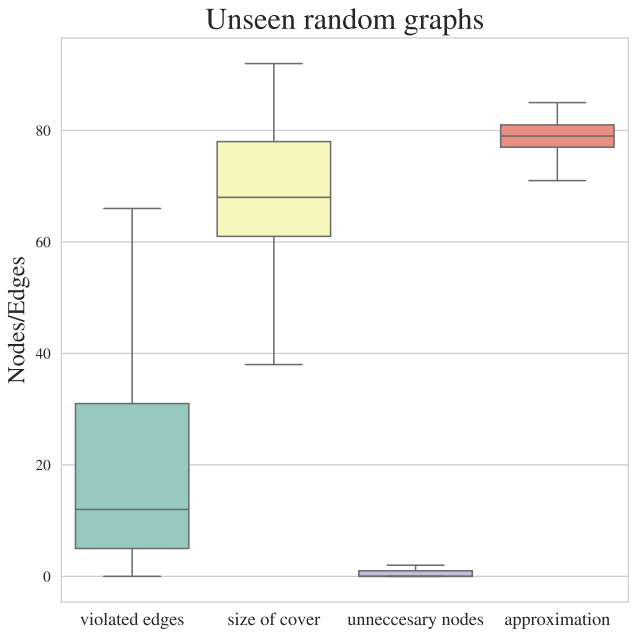

FIG. 5: MVC predictions on 100 random graphs, from training on 100 random graphs.

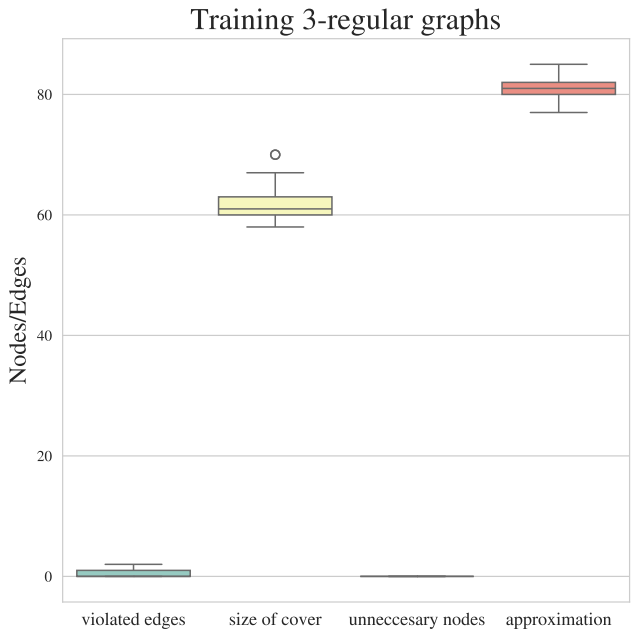

FIG. 7: MVC results from training on 100 3-regular graphs.

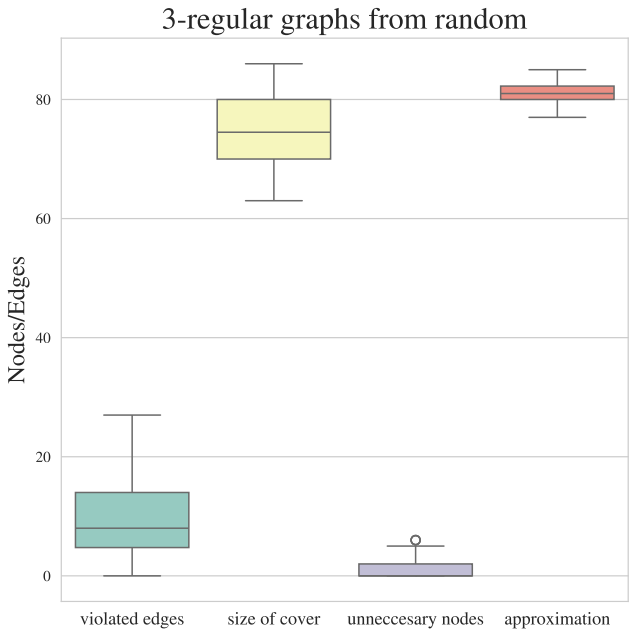

FIG. 6: MVC predictions on 100 3-regular graphs, from training on 100 random graphs.

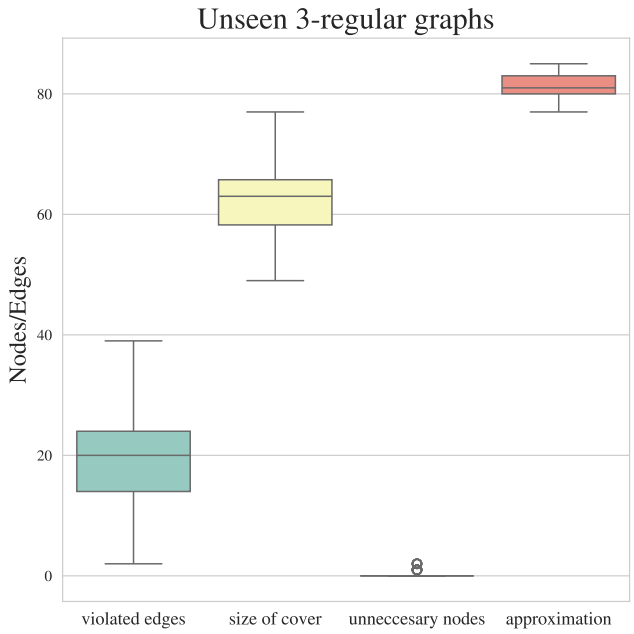

FIG. 8: MVC predictions on 100 3-regular graphs, from training on 100 3-regular graphs.

salesman seem similar, the problems are in practice much less related. When looking for a Hamiltonian cycle, one typically starts with a sparser graph. An example of this could be a social graph of who knows who, where the aim is to pass a message around to each person.

On the other hand, TSP typically concerns itself with complete graphs, or nearly so. As discussed previously, GCNs are not able to function on complete graphs due to oversmoothing. Finding a Hamiltonian cycle in a complete graph is a trivial exercise, simply take an arbitrary ordering of the nodes.

Due to this, the immediate choice of training data

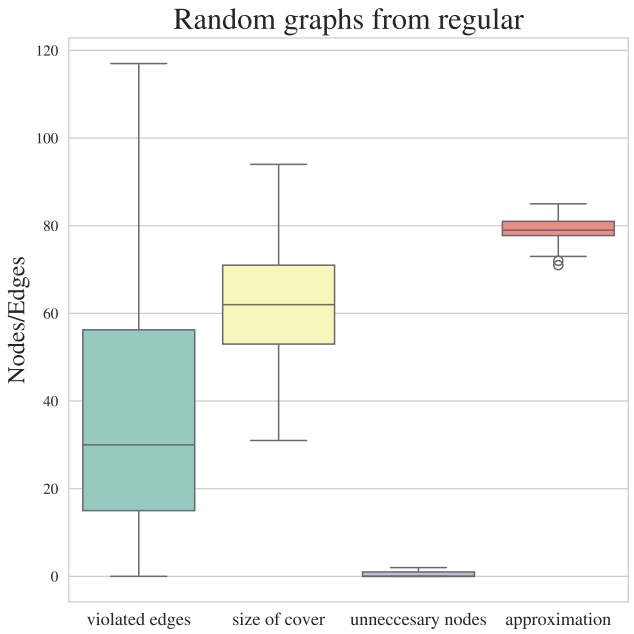

FIG. 9: MVC predictions on 100 random graphs, from training on 100 3-regular graphs.

was rather limited. Because of this, $2N \times 2N$ grids of points was chosen, as it is guaranteed to have Hamiltonian cycles while being relatively sparse (OEIS Foundation Inc., 2024).

The models were trained on grids where nodes are connected to their orthogonal neighbours, as well as grids which were connected to their diagonal neighbours as well. As the orthogonal grids are the ones described previously, and as they contain Hamiltonian cycles while being subgraphs of the diagonal ones, the diagonal graphs are also guaranteed to contain a Hamiltonian cycle.

The models were trained both with and without the TSP term of (16), in order to see if it had a meaningful effect. The models struggled to converged, and therefore consisted of 1 convolution with an MLP of 32 nodes, before using the softmax function in the final prediction layer. After this, the predictions were post processed such that $x_{1,1} = 1$, while $x_{i,1} = x_{1,i} = 0$ for $i = 2, 3, \ldots, n$, in effect fixing that the first node should be the first in the cycle. This should not have a cycle, as it does not matter which node is first.

As we see in Figure 10 and Figure 11, the addition of the TSP term does not seem to have had a meaningful effect. This was a recurring theme across multiple different graphs, as well as with different model architectures.

As we see, the model was not able to predict a valid tour in any case. This is likely to stem from the approach used, as it is attempting to globally predict a route, consisting of n^2 variables. It seems to be improving different line segments at once, running into an issue when they

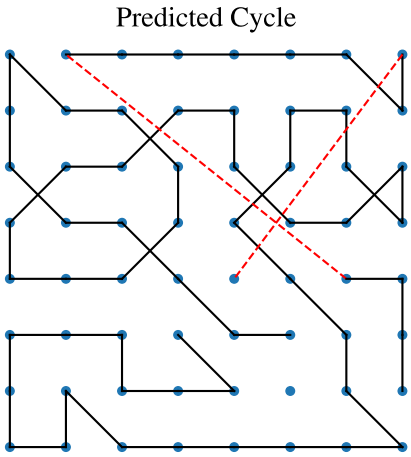

FIG. 10: Model trained on 8×8 diagonal graph, with TSP.

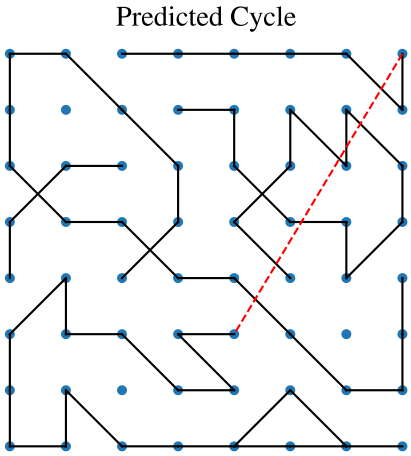

FIG. 11: Model trained on 8×8 diagonal graph, without TSP.

cannot be extended further. This might be due to the fact that the line segments need to be constructed in order, beginning at an index j in the cycle. While the start and end of two lines might lie adjacent visually, if the corresponding indices do not match up, they cannot be connected with this method. In this way, the model falls into a deep local minima. Without major changes to the fundamental framework, I fail to see how this challenge can be overcome.

Joshi *et al.* (2019) used a method not too unlike the one described here. Their method predicts probabilities $x_{u,v}$, denoting the probability that nodes u and v should be connected. They overcame the problem of oversmoothing through utilizing the k-nearest neighbour subgraph, training their network on a labeled data set. As they do not the predicted ordering of the nodes, they decoded

the probabilities using beam search in order to obtain a valid cycle. The use of a labeled set limited their training to graphs with maximally 100 nodes, generalizing poorly with respect to larger graphs. As they note in their paper, the use of heavy post processing might obscure the flaws of the model, which might be akin to what is seen in the models here.

Fellek *et al.* (2024) utilized deep reinforcement learning in combination with an adapted Graph Attention Network (GAT) in their proposed Gated Deep Graph Attention Network (G-DGANet). In this method, the final tour is also constructed iteratively. Their model generalized better for larger graphs, as well as outperforming a number of other machine learning methods.

V. CONCLUSION

Graph Convolution Networks seem promising for a number of combinatorial problems, however it seems likely that they would work better in conjunction with more traditional methods. Without this, one cannot ensure that the problem constraints are satisfied, meaning the predictions are mostly useless.

Using the Ising model framework seems promising for simpler problems like MVC, as each node only predicts a single value. In addition, the effect of the predictions from one node on the loss is only affected by its immediate neighbours. With the TSP formulation, each node effects all others, serving as a complicating factors. It is then not as simple to learn local behaviour.

The poor result on the Traveling Salesman Problem was not entirely surprising, given the complexity of the problem. Tracing its origins back to the 1800s (University of Waterloo, 2022), a lot of effort has gone into solving it efficiently, with no clear solution up to this point.

REFERENCES

M. Aigner (2023), *Discrete Mathematics* (American Mathematical Society).

R. Bar-Yehuda, and S. Even (1985), "A local-ratio theorem for approximating the weighted vertex cover problem," in *Analysis and Design of Algorithms for Combinatorial Problems*, North-Holland Mathematics Studies, Vol. 109, edited by G. Ausiello, and M. Lucertini (North-Holland) pp. 27–45.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang (2018), "JAX: composable transformations of Python+NumPy programs," .

DeepMind, I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, A. Dedieu, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, G. Papamakarios, J. Quan, R. Ring, F. Ruiz, A. Sanchez, L. Sartran, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, M. Stanojević, W. Stokowiec, L. Wang, G. Zhou, and F. Viola (2020), "The DeepMind JAX Ecosystem," .

G. Fellek, A. Farid, S. Fujimura, O. Yoshie, and G. Gebreyesus (2024), "G-dganet: Gated deep graph attention network with reinforcement learning for solving traveling salesman problem," Neurocomputing **579**, 127392.

M. Fredman, and R. Tarjan (1984), "Fibonacci heaps and their uses in improved network optimization algorithms," in *25th Annual Symposium onFoundations of Computer Science, 1984.*, pp. 338–346.

J. Godwin, T. Keck, P. Battaglia, V. Bapst, T. Kipf, Y. Li, K. Stachenfeld, P. Veličković, and A. Sanchez-Gonzalez (2020), "Jraph: A library for graph neural networks in jax." .

I. Goodfellow, Y. Bengio, and A. Courville (2016), *Deep Learning* (MIT Press) `http://www.deeplearningbook.org`.

A. A. Hagberg, D. A. Schult, and P. J. Swart (2008), "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, edited by G. Varoquaux, T. Vaught, and J. Millman (Pasadena, CA USA) pp. 11–15.

D. K. Hammond, P. Vandergheynst, and R. Gribonval (2009), "Wavelets on graphs via spectral graph theory," arXiv:0912.3848 [math.FA].

K. He, X. Zhang, S. Ren, and J. Sun (2015), "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," arXiv:1502.01852 [cs.CV].

J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee (2023), "Flax: A neural network library and ecosystem for JAX," .

C. K. Joshi, T. Laurent, and X. Bresson (2019), "An efficient graph convolutional network technique for the travelling salesman problem," arXiv:1906.01227 [cs.LG].

R. M. Karp (2010), "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, edited by M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey (Springer Berlin Heidelberg, Berlin, Heidelberg) pp. 219–241.

D. P. Kingma, and J. Ba (2017), "Adam: A method for stochastic optimization," arXiv:1412.6980 [cs.LG].

T. N. Kipf, and M. Welling (2017), "Semi-supervised classification with graph convolutional networks," arXiv:1609.02907 [cs.LG].

A. Lucas (2014), "Ising formulations of many NP problems," Frontiers in Physics **2**, 10.3389/fphy.2014.00005.

OEIS Foundation Inc. (2024), "Number of (undirected) hamiltonian cycles on a 2n x 2n square grid of points, entry A003763 in the On-Line Encyclopedia of Integer Sequences," .

A. P. Punnen (2022), *The Quadratic Unconstrained Binary Optimization Problem: Theory, Algorithms, and Applications*, 1st ed. (Springer International Publishing AG, Cham).

M. Raissi, P. Perdikaris, and G. Karniadakis (2019), "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," Journal of Computational Physics **378**, 686–707.

T. K. Rusch, M. M. Bronstein, and S. Mishra (2023), "A survey on oversmoothing in graph neural networks," arXiv:2303.10993 [cs.LG].

M. J. A. Schuetz, J. K. Brubaker, and H. G. Katzgraber (2022), "Combinatorial optimization with physics-inspired graph neural networks," Nature Machine Intelligence **4** (4),

367–377.
S. Singh (2020), "The ising model: Brief introduction and its application," .
University of Waterloo (2022), "History of the tsp," .
Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu (2021), "A comprehensive survey on graph neural networks," IEEE Transactions on Neural Networks and Learning Systems **32** (1), 4–24.