LEARNING TO SOLVE THE HIDDEN CLIQUE PROBLEM WITH GNNS

Anonymous authors
Paper under double-blind review

ABSTRACT

We study data-driven methods for the hidden clique problem in random graphs. The training data is obtained by hiding a clique in the random graph, where the signal to noise ratio is tuned by choosing the size of the hidden clique and the density of the random graph. Using synthetic datasets allows us to test empirically the performance and generalization properties of various graph neural network (GNN) architectures at different levels of difficulties for the task. We compare message passing GNNs and GNNs augmented with a single quadratic operation (matrix multiplication) first introduced in (Maron et al., 2019). Adding skip connections and normalization to these augmented GNNs is shown to improve their learning process and their generalization properties without any loss in time complexity. For hard instances of our hidden clique problem, they are shown to outperform message passing GNNs.

1 INTRODUCTION

The domain of Graph Neural Networks (GNNs) has seen huge development in recent years. Their ability to tackle a wide range of problems using graphs has attracted many researchers aiming at bettering the field. Indeed this common data structure can be used to represent many useful interactions ranging from social networks (Leskovec & Krevl, 2014) to music recommendation (Dolgikh & Jelinek, 2015), to biological structures or molecules (Kersting et al., 2016) and many other uses.

The community has been very active, developing so many architectures that we now even have taxonomies of GNNs (Chami et al., 2020). However, there hasn't been a lot of attention to testing and benchmarking these models in all domains of graphs until very recently (Hu et al., 2021; Palowitch et al., 2022). Indeed, we have only a handful of test data sets on which models are tested. This can have many drawbacks on the advancement of GNNs. People trying to use a neural network might choose a network not fit for their task because it is considered better in the literature. We might also overfit on the data sets we are using (Recht et al., 2018). It could also be worse, we might not even create useful GNNs, a thesis supported by recent articles (Dacrema et al., 2019; Palowitch et al., 2022) showing that some baselines seem to give the same performances as our complex models.

Our first goal was to study the training of GNNs on hard combinatorial problems, in our case, we set sights on the maximum clique problem. However, as it is a NP-problem (Karp, 1976), it is computationally impossible to obtain a lot of labeled data. To avoid this obstacle, different methods have been designed (Cappart et al., 2021). One of them is training GNNs with already solved instances, for them to give an expected good baseline for heuristic-based algorithms (Li et al., 2018). This still requires exact solving for training the GNN, and for this reason can't be scaled much further than exact solvers. Another method is bypassing data labeling by using reinforcement learning (RL) (Li et al., 2021). The downside is that the computational cost stays huge and a lot of engineering must be done on hyper-parameters for RL to succeed. Our procedure is inspired by previous works done in statistical physics (Moharrami et al., 2019). The idea is to solve a relaxed problem, where the solution has been hidden (or planted) by our means. The problem is thus simpler and requires no computation to solve, but it was proven that insight can be gained on the original problem. In our case, the hidden clique problem is the relaxed version of the maximum clique problem. However, training on the HCP is much easier as we can use a simple, well-studied, supervised learning pipeline.

We study here the HCP, a problem with real-world applications, notably in biology (Yang et al., 2014). In a graph, a clique is a complete subgraph, that is, a subset of nodes which are all connected. The problem here is to retrieve a previously planted clique in a random graph. The size of the planted clique can be tuned as a signal-to-noise ratio, to test GNNs on its different regimes of hardness that we present in section 3.1.

We compare the performances of various GNNs on this task, namely GatedGCN (Bresson & Laurent, 2017), GAT (Veličković et al., 2017), GIN (Xu et al., 2018) and our residual folklore graph neural network (RSFGNN). The starting point for this GNN, first introduced in Maron et al. (2019), was a design of GNN augmented with a quadratic operation to have better discerning power for the Weisfeiler-Lehman isomorphism test (Leman & Weisfeiler, 1968). It was proven that these kinds of GNNs are indeed more powerful than message-passing GNNs (MGNNs) in Azizian & Lelarge (2020). We have modified this base architecture slightly: we haveadded normalization to prevent explosion of values when using an FGNN architecture on bigger graphs, making it mostly agnostic to the size of the input graph. We have also added residuals (He et al., 2016), to make the architecture more stable with more layers. In this article we advocate that the RSFGNN, while often put aside because of its space and time complexity, can actually prove very useful on small to medium sized graphs, especially when they are dense, as it can perform better than MGNNs and even faster on certain sets of data.

2 EQUIVARIANT GRAPH NEURAL LAYERS

In this section, we describe our GNN layer which is adapted from Maron et al. (2019). We use notations similar to Dwivedi et al. (2020) or Azizian & Lelarge (2020).

Notation: we denote by $\mathbb{F}, \mathbb{F}_0, \mathbb{F}_{1/2}, \mathbb{F}_1, \dots$ arbitrary finite-dimensional spaces of the form $\mathbb{R}^p$ (for various values of p) typically representing the space of features. Products of vectors in $\mathbb{R}^p$ always refer to component-wise products. Let $[n] = \{1, \dots, n\}$. Graphs are typically represented by their discrete structure with an additional feature vector on each vertex: the discrete graph $G = (V, E)$ with n nodes $V = [n]$ and edges $E \subseteq V^2$ (with no weights on edges) and with a vector $h^0 \in \mathbb{F}^n$ representing features on the vertices.

2.1 MESSAGE PASSING GRAPH LAYERS

In such a setting, Message passing GNN (MGNN) are defined inductively as follows: let $h_i^\ell \in \mathbb{F}_\ell$ denote the feature at layer ℓ associated with node i, the updated features $h_i^{\ell+1}$ are obtained as: $h_i^{\ell+1} = f\left(h_i^\ell, \{\{h_j^\ell\}\}_{j\sim i}\right)$, where $j \sim i$ means that nodes j and i are neighbors in the graph G, i.e. $(i, j) \in E$, and the function f is a learnable function taking as input the feature vector of the center vertex h_i^ℓ and the multiset of features of the neighboring vertices $\{\{h_j^\ell\}\}_{j\sim i}$. In other words, the function f is invariant with respect to $(h_j^\ell)_{j\sim i}$. The invariance of f ensures that the mapping $(h_i^\ell)_{i\in[n]} \mapsto (h_i^{\ell+1})_{i\in[n]}$ is equivariant. Indeed, it follows from Zaheer et al. (2017), that any such function f can be approximated by a layer of the form,

$$h_i^{\ell+1} = f_0\left(h_i^\ell, \sum_{j\sim i} f_1\left(h_j^\ell\right)\right), \tag{1}$$

where $f_0 : \mathbb{F}_\ell \times \mathbb{F}_{\ell+1/2} \to \mathbb{F}_{\ell+1}$ and $f_1 : \mathbb{F}_\ell \to \mathbb{F}_{\ell+1/2}$, so that $\mathbb{F}_\ell$ is the feature space for the ℓ-th layer. Message passing layers are very useful and efficient for sparse graphs. Many variants have been proposed with different choices for the parametrization of the functions f_0 and f_1 above. In our experiments, we used three different architectures: GatedGCN (Bresson & Laurent, 2017), GAT (Veličković et al., 2017) and GIN (Leskovec & Jegelka, 2019). We refer to (Dwivedi et al., 2020) for an unified presentation of their respective architectures.

2.2 GRAPH LAYERS WITH TENSORS OF ORDER 2

Maron et al. (2019) proposed an alternative GNN architecture where now graphs are seen as tensors of order 2: $G \in \mathbb{F}^{n^2}$. The classical representation of a graph by its (weighted) adjacency matrix is a

tensor of order 2 in $\mathbb{R}^{n^2}$. Note that features on vertices can be encoded on the diagonal terms. To be consistent with the notation introduced for MGNN, we will denote by $\mathbf{h} = (h_{i\to j})_{i,j\in[n]}$ a tensor of order 2 where $h_{i\to j} \in \mathbb{F}$ is interpreted as a feature or hidden state associated with the directed pair of vertices (i, j). Note that (i, j) does not need to be an edge of the graph: we can have features for 'non-edges'.

We can encode a graph $G = (V, E)$ into a tensor $(h^0_{i\to j})_{i,j\in[n]}$ as follows: if $(i, j) \notin E$, we set $h^0_{i\to j} = 0$, if $(i, j) \in E$, we set $h^0_{i\to j} = 1$ (and if node i has a features, we encode it in $h^0_{i\to i}$). The following layer was first introduced in Maron et al. (2019) (see Block structure in their Section 6) inspired by the Folklore Weisfeiler-Lehman test and called Folklore Graph Layer (FGL) in Azizian & Lelarge (2020)

$$h^{\ell+1}_{i\to j} = f_0\left(h^\ell_{i\to j}, \sum_{k\in[n]} f_1\left(h^\ell_{i\to k}\right) f_2\left(h^\ell_{k\to j}\right)\right), \tag{2}$$

where $f_0 : \mathbb{F}_\ell \times \mathbb{F}_{\ell+1/2} \to \mathbb{F}_{\ell+1}$ and $f_1, f_2 : \mathbb{F}_\ell \to \mathbb{F}_{\ell+1/2}$ and the product of vectors in $\mathbb{F}_{\ell+1/2}$ is done component-wise. In Maron et al. (2019), each f_0, f_1, f_2 is modeled by a MLP. We see that the $\sum_k$ term in equation 2 needs to be computed for all pairs i, j so that indeed this is a matrix multiplication between the matrices $f_1(\mathbf{h}^\ell)$ and $f_2(\mathbf{h}^\ell)$ and this can be computed efficiently as a convolution with a kernel of size one. It is shown that GNNs obtained by stacking FGLs have the same separating power as the Folklore Weisfeiler-Lehman test in Maron et al. (2019) and better approximation power than MGNN in Azizian & Lelarge (2020).

In this paper, we modify slightly the architectures proposed in Maron et al. (2019) as follows:

- we use **normalization** Ba et al. (2016). More precisely, let denote by $m_{i\to j}$ resulting form the sum over k inside equation 2, i.e.

  $$m_{i\to j} = \sum_{k\in[n]} f_1\left(h^\ell_{i\to k}\right) f_2\left(h^\ell_{k\to j}\right)$$

  We have for each $i, j \in [n]$, $m_{i\to j} \in \mathbb{F}_{\ell+1/2} = \mathbb{R}^d$ which is a sum of n terms, hence we normalize it such that $\sum_{\alpha\in[d]} m^2_{i\to j}(\alpha) \approx 1$. We use a similar normalization inside the function f_0 so that $h^{\ell+1}_{i\to j}$ in equation 2 is also normalized.
- we use **residual connections** He et al. (2016) so that equation 2 becomes:

  $$h^{\ell+1}_{i\to j} = h^\ell_{i\to j} + f_0\left(h^\ell_{i\to j}, \sum_{k\in[n]} f_1\left(h^\ell_{i\to k}\right) f_2\left(h^\ell_{k\to j}\right)\right), \tag{3}$$

  where the output of f_0 is normalized.

As in Maron et al. (2019); Azizian & Lelarge (2020), the last layer of the GNN is a projection taking as input a tensor of order 2 and producing as output a tensor in $\mathbb{F}^n$ where each component is associated to a node of the original graph.

Note that using normalization for FGNNs is already proposed in Dwivedi et al. (2020) (see their paragraph: Underlying challenges for training WL-GNNs) as a way to get more stable training for FGNNs. In Dwivedi et al. (2020), the authors "observe relatively high standard deviation in the performance" of FGNNs. They "experimented with layer normalization but without success.". In our RSFGNN, we managed to add both layer normalization and residual layers to stabilize our architecture.

2.3 Theoretical comparison between MGNNs and FGNNs

Here we summarize known theoretical results about MGNNs and FGNNs. First, note that normalization and skip connections introduced in RSFGNNs do not change the general formalism of equation equation 2 so that the theoretical results from Maron et al. (2019); Azizian & Lelarge (2020) are still valid in our case (we will use FGNNs as a term regrouping classical FGNNs and our RSFGNNs in this part): FGNNs obtained by stacking FGLs have the same separating power as the Folklore

Weisfeiler-Lehman test Maron et al. (2019). As a result, we see that the separating power of MGNNs is the same as the 2-WL algorithm and the separating power of FGNNs is the same as the 3-WL algorithm. As a result, FGNNs have better approximation power than MGNN. Connecting GNN with the Weisfeiler-Lehman test was first done in Leskovec & Jegelka (2019) and a survey of these results with connection with the literature can be found in Azizian & Lelarge (2020) or Geerts & Reutter (2022).

This better expressiveness of FGNNs compared to MGNNs comes at a cost. For FGNNs, the space complexity is $O(n^2)$ and the time complexity is $O(n^3)$ because a dense matrix multiplication is required (even if the graph is sparse). For MGNNs, the space and time complexity is typically $O(E)$, so that for sparse graphs this can become as small as $O(n)$. This is a well-known limitation of FGNNs preventing their use for large-scale graphs, see for example Dwivedi et al. (2020). In this paper, we consider only small to medium sizes graphs for which FGNNs are still easy to train. As explained in the introduction, this setting is still relevant in various problems. Focusing our interest on medium size graphs allows us to consider hard tasks (see below for more details about the hidden clique problem). In this setting, the search space in which MGNNs live is much smaller than the search space of FGNNs as MGNNs have a time complexity at most $O(n^2)$ whereas FGNNs can have a time complexity of $O(n^3)$. In other words, with MGNNs, we restrict ourselves to quadratic algorithms whereas with FGNNs, we consider cubic algorithms. This naturally raises the question to know whether FGNNs are actually able to perform better than MGNNs through supervised learning.

3 Experimental methods

As explained above, we are mostly interested in the impact of various GNN architectures in a supervised setting. FGNNs have better theoretical guarantees than MGNNs, yet in practice, their performances appear limited. In Dwivedi et al. (2020), the authors write: "GCNs outperform WL-GNNs on the proposed datasets." In their experiments, even on medium-scale datasets, GCNs outperform the GNNs proposed in Maron et al. (2019) and these GNNs "face loss divergence and/or out-of-memory errors when trying to build deeper networks."

In this section, we explain our dataset generation and the learning procedure.

3.1 Dataset generation with planted clique

Basic notations Let $G(n,p)$ denote the random graph on n labeled vertices obtained by choosing, randomly and independently, every pair ij of vertices to be an edge with probability p. Let $G(n,p,k)$ denote the probability space whose members are generated by choosing a random graph $G(n,p)$ and then by planting randomly a clique of size k in it, which we'll nae the hidden/planted clique. We denote by $\omega(G)$ the size of the maximum clique in G.

Complexity of the problem For G from the distribution $G(n,p)$, it is known that with high probability, the value of $\omega(G)$ is either $\lceil r(n) \rceil$ or $\lfloor r(n) \rfloor$, for a certain function $r(n) = (2 + o(1)) \ln_{1/p} n$ which can be written explicitly Alon & Spencer (2004). It was shown by Grimmett & McDiarmid (1975) that, for random graphs of $G(n,p)$, when $n \to \infty$, the size of the maximum clique is, with probability one:

$$2\log_{1/p}(n) + o(\log_{1/p}(n)) \tag{4}$$

In most of our article, we use $p = 1/2$:

$$2\log_2(n) + o(\log_2(n))$$

For a graph G from the distribution $G(n, 1/2, k)$:

- if $k \gg \ln n$, there is a unique clique of size k in G with high probability
- if $k \ll \ln n$, there are $n^{\Omega(\ln n)}$ cliques of size larger than $(2-\epsilon)\ln_2 n$.

In particular, we see that as soon as $k \gg \ln n$, the maximum clique is indeed the hidden one, whereas if $k \ll \ln n$, the hidden clique is not the maximum clique.

We can see k as a signal-to-noise ratio (SNR) : $k \ll \omega(G)$ means a hidden small clique, drowned in many cliques of the same size, thus a small SNR; $k \gg \omega(G)$ means a huge hidden clique, easy to find, thus a large SNR.

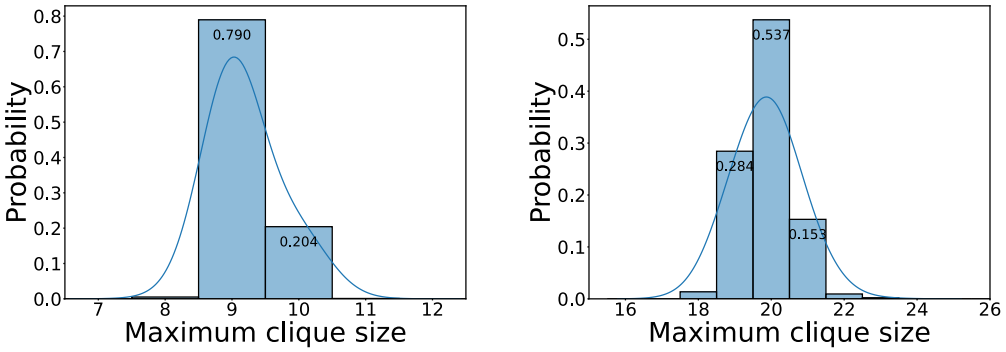

Figure 1: Plot showing the empirical size of the maximum cliques over 960 Erdős-Rényi graphs of size $N = 100$. The edge probabilities are $p = 0.5$ on the left, and $p = 0.8$ on the right. We can see that the maximums are different than the limit presented in equation 4 which are $\omega_{p=0.5}(G) \approx 13$ and $\omega_{p=0.8}(G) \approx 41$, as our graphs are quite small. We obtain empirical peaks at around $\hat{\omega}_{p=0.5}(G) \approx 9$ and $\hat{\omega}_{p=0.8}(G) \approx 20$. We computed these values using a parallel maximum clique algorithm provided by Rossi et al. (2013)

.

Implementation In this paper, we treat HCP as a mainstream node classification problem: we label the nodes as a positive class if they are part of the hidden clique. Another way of presenting the problem would be to classify edges, labelling them as positive if they are linking two nodes from the hidden clique. Even though our RSFGNN can be easily switched between these two kinds of tasks, we discard edge-classification, as we want to compare our architecture to others that were designed more easily for node classification.

Data generation We generate a random Erdős-Rényi graph of size N with edge probability p. Afterwards, we chose a subset S of k nodes in this graph and add all edges needed to make a clique with these edges. It is possible that a subset S' forming a clique containing all nodes of S exists, but we decided not to check for such cliques to better control our SNR. As such, the SNR is only dependant on k.

Parameters In most of our experiments, we used $N = 100$ nodes and the probability of edges $p = 1/2$ (to place ourselves in the dense regime) for all the graphs in our data sets. Using equation 4, we can calculate an estimated clique size of $\omega(G) \approx 13$. In practice, as our graphs are not large, we can solve the MCP exactly using the Bron-Kerbosch enumeration algorithm. Empirically, we find a maximum clique of size $\omega(G) \approx 9$. In a second part, we compared architectures on even denser graphs: $N = 100$ and $p = 0.8$ to see how each model reacted. The estimate of $\omega(G) \approx 41$ is much farther than the empirical clique size (calculated over 960 graphs) of $\omega(G) \approx 20$. This empirical data is shown in figure 1.

3.2 Models

We chose 3 other models for comparison in our experiments :

- **GatedGCN** (Bresson & Laurent, 2017): a GNN that uses residual connections, batch normalization and edge gates. It was chosen in spite of the simple GCN (Kipf & Welling, 2016) (that showed poor performances on the dataset presented here), because it showed good results on recent benchmark papers (Dwivedi et al., 2020).
- **GAT** (Veličković et al., 2017): a model of graph attention. Attention mechanisms have already showed their potential in other domains of machine learning, especially with the rise of transformers (Vaswani et al., 2017). We wanted this architecture to be represented.
- **GIN** (Leskovec & Jegelka, 2019): another architecture based on the Weisfeiler-Lehman Isomorphism test (Leman & Weisfeiler, 1968), designed to have a better discerning power between graphs than GCN or GraphSAGE (Hamilton et al., 2017).

Table 1: Hyperparameter table for our different models

Model	In feats	Hidden feats	Out feats	#Layers	MLP depth	#Heads
GIN	1	64	2	3	3	N/A
GatedGCN	1	64	2	3	3	N/A
GAT	1	8	2	2	3	8
RSFGNN (Ours)	2	64	1	3	3	N/A

We used these models to present a small idea of the current state of the GNN models. We implemented them using the reference from the DGL library [1] (Wang et al., 2019). The RSFGNN model was implemented starting from the model given by Azizian & Lelarge (2020). All models use PyTorch (Paszke et al., 2019) as a framework with the Pytorch Lightning wrapper (Falcon, 2019).

3.3 Training and testing procedure

We created 16 data sets for HCP of graphs of size $N = 100$, with a hidden clique size ranging from 5 to 20 (chosen to go from low-signal to high-signal). We trained 16 instances of our 4 models (RSFGNN, GatedGCN, GAT, GIN) each on one of the 16 dataset. Then for each of these instances, we test them on all the datasets separately, recording each time the F_1-score, AUC score and other metrics, like size of the clique after a beamsearch (for further details, see the supplementary material. We have also provided a simple baseline for comparison. It simply ranks the nodes in a descending degree order (higher degree is more probable to be in the clique). All of these models output a scalar for each node, that can be compehended as the a probability for this node to be a part of the hidden clique. We then calculate the metrics accordingly.

Hyperparameters The hyper-parameters for the models were derived from the references, and manually tweaked a little to get better performances for each model. They are listed in table 1.

Experimental setup We use the pytorch provided Adam optimizer with its base parameters. We train our models for 100 epochs, using a learning rate of 0.001. A scheduler reduces the learning rate by two if the GNN hasn't gotten better in 3 epochs. If the learning rate goes under 10^{-7}, the learning is stopped early. The training set is comprised of 10000 graphs of size $N = 100$, the validation set of 1000 graphs of size $N = 100$ and the test set of 960 graphs of size $N = 100$. This last set has 960 graphs to evenly split the data among different CPUs (in our case 32) when computing some non-parallelizable metrics such as beam-search.

4 Results

4.1 Computational time analysis

One of the possible uses we have cited is the ability of GNNs to approximate solutions of problems we can't solve exactly, for example, the MCP. However, for this to be useful, we need them to compute faster than the exact solver. As our graphs are medium-sized, we verified it was the case. We compared the time taken for computing a GNN output against an exact and parallel solver for maximum clique problem (PMC) (Rossi et al., 2013) on differently-sized graphs. These result are compiled in table 2. We can observe that using GNNs indeed has an overwhelming advantage over using exact solving, even with small-sized graphs. GNNs are 10 times faster at least even for graphs or size $N = 100$ and more than 100 times for graphs of size $N = 800$. The difference gets even wider when using denser graphs, where our exact solver failed.

We can also see that in practice, even though RSFGNNs implement a dense quadratic computation with a theoretical $O(n^3)$ time complexity, they still run faster in python code than GIN and GatedGCN. These two models use the Deep Graph Library, which is made for computing on sparse graphs, it

[1] DGL reference: `https://github.com/dmlc/dgl/tree/master/examples/pytorch`

Table 2: Computational time (in seconds/graph). For the GNNs, we have used 10000 (resp. 960) samples for $N = 100$ (resp. $N = 800$) on an Nvidia GTX 1080Ti. For the parallel maximum clique algorithm (PMC), we have used 10000 (resp. 100) samples for $N = 100$ (resp. $N = 800$) on Intel Xeon 5218 for a total of 64 cores. For $N = 800$, $p = 0.8$ the PMC has not managed to solve one instance before a 7 days time-out limit. Some values of standard deviation are big because of fluctuation of charge on our processing units.

Model	$N = 100$		$N = 800$	
	$p = 0.5$	$p = 0.8$	$p = 0.5$	$p = 0.8$
RSFGNN	0.0044 ± 0.001	0.0044 ± 0.001	0.007 ± 0.006	0.007 ± 0.002
GIN	0.0047 ± 0.001	0.0047 ± 0.001	0.012 ± 0.023	0.017 ± 0.032
GatedGCN	0.0064 ± 0.012	0.0064 ± 0.012	0.012 ± 0.004	0.031 ± 0.005
PMC	0.0725 ± 0.008	0.1018 ± 0.011	3.479 ± 0.164	OOT (7 days)

might not be optimized in the case of dense ones, contrary to our RSFGNN that does dense matrix multiplications.

These results justify the study of how GNNs behave on dense graphs. They show that in the case where a practitioner has to compute lots of instances of a problem, and does not need an exact solution, GNNs are indeed a good fit. They also show that when dense graphs are in play, a more expressive approach through the use of RSFGNNs should be preferred as is not even costly in computational time.

4.2 Results on the architectures

Firstly, we have found that some architectures are hard to train on dense graphs: the GATs training hardly converged on our HCP. Even though we haven't had the time to tweak the learning procedure for them, it is surprising as the code for GATs was taken directly from the DGL library, like for GIN and GatedGCN. This is probably due to the dense nature of the data set, having a ratio of number of edges on number of nodes of around 50. Even the GAT architectures that have managed to learn also seem to be less successful than the other structures. We have provided their scores in the supplementary material. GATs may be a bad fit for learning on dense graphs, and should probably be avoided for practitioners in this regime.

One other finding is that there is a change of order between some architectures at different points of training and testing, namely, GIN trained at clique size 11 is always better than the GatedGCN, but it is the other way when training at clique size 20. This confirms that practitioners should be careful when choosing their architectures. GatedGCNs seem to not be too dependant on what clique size they have learnt, while GINs seem to correspond best to the ideas presented in section 3.1: they perform better in the intermediate regime, when the problem is not too hard nor too easy.

Most of all, we can see that the RSFGNN performs better than any other architecture in this dense regime. We also observe that the RSFGNNs trained at a planted clique size of 10 and 11 are performing the best. This makes sense when comparing to the maximum clique size (see figure 1): it is the regime predicted in section 3.1 between the impossible retrieval phase (planted clique smaller than the maximum clique) and the easy regime (planted clique bigger than the maximum clique). In this regime, the RSFGNN can latch onto the signal and still do good when the planted size is a bit smaller. The RSFGNN shows much better metrics than the MGNNs.

To add robustness on this hypothesis, we tested the same neural networks, this time on random Erdős-Rényi graphs of size $N = 100$ and edge probability $p = 0.8$. We trained them directly at empirical the limit between the easy and the hard phase, at a clique size of 21 (see figure 1). We can also observe that RSFGNN does much better than the other two in figure 3, which was to be expected as we only chose denser graphs.

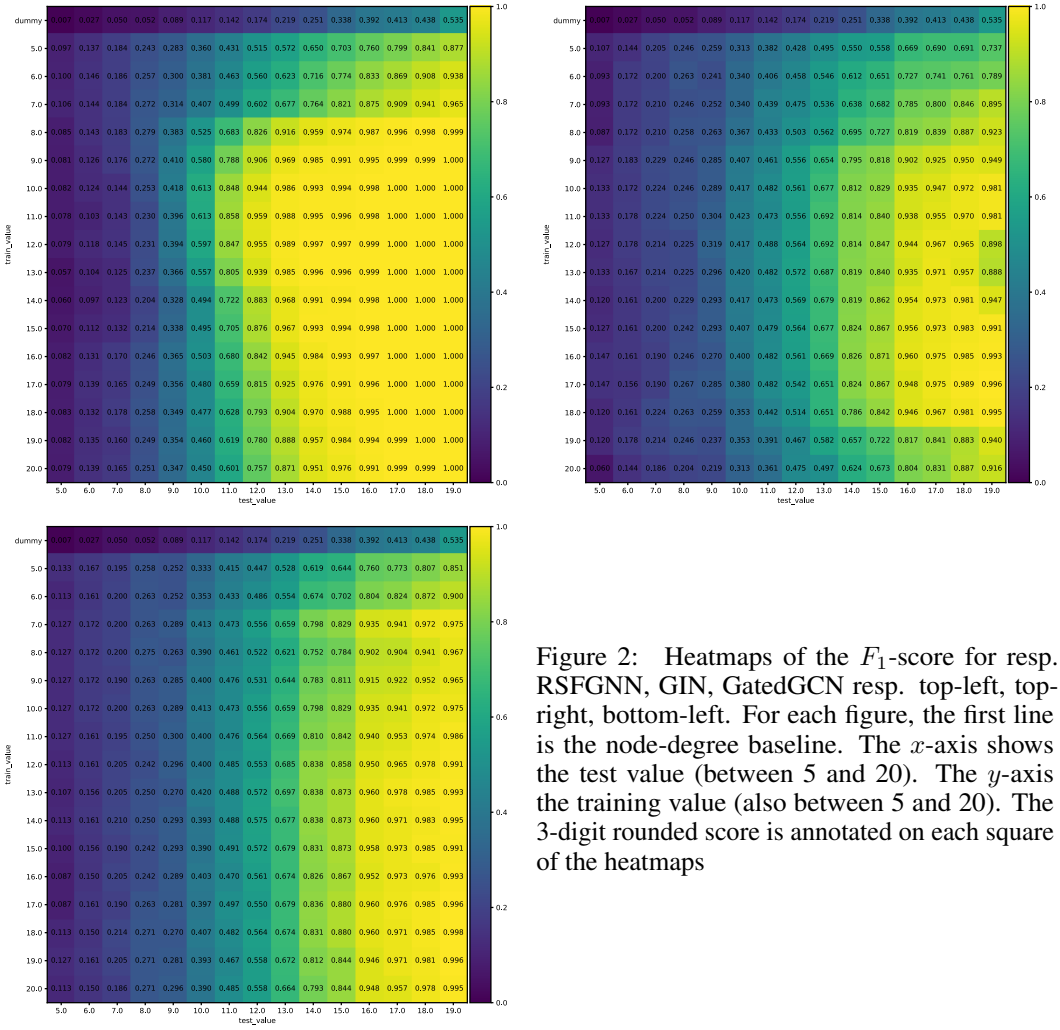

Figure 2: Heatmaps of the F_1-score for resp. RSFGNN, GIN, GatedGCN resp. top-left, top-right, bottom-left. For each figure, the first line is the node-degree baseline. The x-axis shows the test value (between 5 and 20). The y-axis the training value (also between 5 and 20). The 3-digit rounded score is annotated on each square of the heatmaps

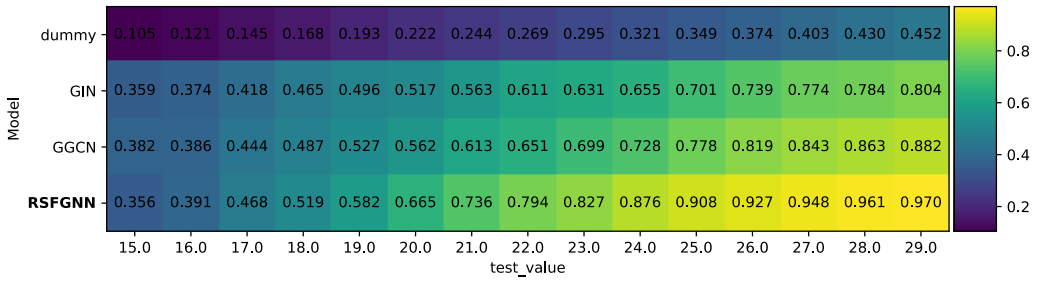

Figure 3: Heatmap showing the F1-score performance of the baseline, GIN, GatedGCN and RSFGNN over random graphs of size $N = 100$ and edge probability $p = 0.8$ depending on the hidden clique size (on the x-axis). We can observe that the RSFGNN does best at nearly all points.

5 CONCLUSION

In this paper, we presented our method for solving the hidden clique problem by the means of a simple pipeline using supervised learning with graph neural networks. By tuning the parameters of the data sets, namely the size of the hidden clique and the density of our random graphs, we showed the regions where the GNNs can provide the most useful insights. We compared different GNNs where they are rarely tested: on dense random graphs, justifying these tests by exhibiting different uses in real-world applications. We compared state-of-the-art message-passing GNNs to the RSFGNN, a GNN augmented by a quadratic operation, to which we added residuals and normalization to stabilize its learning with more layers. We confirmed empirically the theory of Azizian & Lelarge (2020) that they can be more expressive than regular MGNNs, when they are used on the right domain. We showed that they might be the best choice for working on small to medium sized graphs, especially if they are dense. Indeed, in this regime, MGNNs also have a quadratic space complexity, and as they are usually optimized for sparse graphs, RSFGNNs also compute instances faster than them.

REFERENCES

Noga Alon and Joel H Spencer. *The probabilistic method*. John Wiley & Sons, 2004.

Waïss Azizian and Marc Lelarge. Characterizing the expressive power of invariant and equivariant graph neural networks. *CoRR*, abs/2006.15646, 2020. URL https://arxiv.org/abs/2006.15646.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017. URL http://arxiv.org/abs/1711.07553.

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks, 2021. URL https://arxiv.org/abs/2102.09544.

Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy, 2020. URL https://arxiv.org/abs/2005.03675.

Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. Are we really making much progress? a worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems*. ACM, sep 2019. doi: 10.1145/3298689.3347058. URL https://doi.org/10.1145%2F3298689.3347058.

Dmitry Dolgikh and Ivan Jelinek. Graph-based music recommendation approach using social network analysis and community detection method. pp. 221–227, 06 2015. doi: 10.1145/2812428.2812453.

Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *CoRR*, abs/2003.00982, 2020. URL https://arxiv.org/abs/2003.00982.

The Pytorch Team Falcon. Pytorch lightning, 2019. URL https://www.pytorchlightning.ai.

Floris Geerts and Juan L Reutter. Expressiveness and approximation properties of graph neural networks. *arXiv preprint arXiv:2204.04661*, 2022.

G. R. Grimmett and C. J. H. McDiarmid. On colouring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77(2):313–324, 1975. doi: 10.1017/S0305004100051124.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2017. URL https://arxiv.org/abs/1706.02216.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/db8e1af0cb3aca1ae2d0018624204529-Paper-round2.pdf.

R Karp. Probabilistic analysis of some combinatorial search problems. traub, jf (ed.): Algorithms and complexity: New directions and recent results, 1976.

Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016. URL http://graphkernels.cs.tu-dortmund.de.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016. URL https://arxiv.org/abs/1609.02907.

AA Leman and Boris Weisfeiler. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsiya*, 2(9):12–16, 1968.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

Keyulu Xu Weihua Hu Jure Leskovec and Stefanie Jegelka. How powerful are graph neural networks. *ICLR*, 2019.

Kaiwen Li, Tao Zhang, Rui Wang, Yuheng Wang, Yi Han, and Ling Wang. Deep reinforcement learning for combinatorial optimization: Covering salesman problems. *IEEE Transactions on Cybernetics*, pp. 1–14, 2021. doi: 10.1109/tcyb.2021.3103811. URL https://doi.org/10.1109%2Ftcyb.2021.3103811.

Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search, 2018. URL https://arxiv.org/abs/1810.10659.

Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks, 2019. URL https://arxiv.org/abs/1905.11136.

Mehrdad Moharrami, Cristopher Moore, and Jiaming Xu. The planted matching problem: Phase transitions and exact results, 2019. URL https://arxiv.org/abs/1912.08880.

John Palowitch, Anton Tsitsulin, Brandon Mayer, and Bryan Perozzi. Graphworld: Fake graphs bring real insights for gnns, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do cifar-10 classifiers generalize to cifar-10?, 2018. URL https://arxiv.org/abs/1806.00451.

Ryan A. Rossi, David F. Gleich, Assefaw H. Gebremedhin, and Md. Mostofa Ali Patwary. Parallel maximum clique algorithms with applications to network analysis and storage, 2013. URL https://arxiv.org/abs/1302.6256.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017. URL https://arxiv.org/abs/1710.10903.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2018. URL https://arxiv.org/abs/1810.00826.

Lei Yang, Xudong Zhao, and Xianglong Tang. Predicting disease-related proteins based on clique backbone in protein-protein interaction network. *International journal of biological sciences*, 10: 677–88, 06 2014. doi: 10.7150/ijbs.8430.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. *arXiv preprint arXiv:1703.06114*, 2017.

A DETAILS ON THE METRICS USED

We here present more thoroughly the different metrics we used in this paper and how we implemented them.

F_1-score For this metric, we 'cheat' by knowing how large the hidden clique should be. If we are testing on a graph with a hidden clique size k, we keep the set of the k most probable nodes to evaluate the F_1-score. We then define the *precision* as the proportion of nodes in this set that actually belong to the hidden clique, and the *recall* as the proportion of nodes retrieved from the hidden clique compared to the total number of nodes of the hidden clique. The F_1-score is then defined as :

$$F_1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

We chose this metric as it is the most commonly used one for such types of problems.

AUC score This score doesn't rely on knowing anything about the solution of the HCP. To define the AUC score, we first define:

	Positive class	**Negative class**
Selected by model	True Positive (TP)	False Positive (FP)
Not selected by model	False Negative (FN)	True Negative (TN)

Then, we can define the true positive rate $\text{TPR} = \text{TP}/(\text{TP} + \text{FN})$ and the false positive rate $\text{FPR} = \text{FP}/(\text{FP} + \text{TN})$. We know the models outputs a probability p_i for each node i to be part of the hidden clique. We define a threshold $\alpha \in [0, 1]$. A node i will then be labeled as positive class if and only if $p_i > \alpha$.

We can then trace the receiver operating characteristic (ROC) curve of $\text{TPR}(\alpha)$ against $\text{FPR}(\alpha)$. Then, we can compute the area under this curve, which is the AUC. A perfect model will have an AUC of 1.

We chose the AUC score to be a more objective score that can easily be used in any other classification problem. As we saw, it's very similar to the F1 in fine.

Beam search for max clique As we have discussed, it can be interesting to test how our HCP can translate to a MCP. We have added a beam search to form a clique from the unstructured output of our GNNs. For the inference phase, once we obtain the output probability matrix, for each node, we sum the values of its edges, yielding us what we will call their *inferred degree*. Our algorithm then proceeds as follows:

1. Order the nodes by their inferred degree, in descending order, let's call them $v_1, v_2, \ldots, v_n$
2. Create the container list that will store the cliques, with a maximum beam size b (in our case, it ranges between $b = 1$ (greedy search) to $b = 1280$), meaning we will keep in memory a maximum of b cliques. This list will always be ordered by descending clique size and descending degree (with the degree of a clique defined as the sum of the degrees of each node in the clique)
3. Initialize the container with the clique of only the most probable node v_1
4. For each node v_k from $k = 2$ to $k = n$:
   (a) For each clique stored in the container, check if adding v_k to it still gives a clique, and if so insert the newly constructed clique in the container.
   (b) If it's the inference phase, add the clique containing only v_k
   (c) If the container is bigger than the beam size b, truncate it
5. Keep the first element of the container, as it will be the biggest clique found

Other possible metrics There are many possible metrics that could be used depending on the application, for instance, any F_β score, creating an asymmetry between precision and recall for the computation of the F-score. We also explored using a beam-search (a popular method for getting structured data out of a neural network) to create real cliques from the output of our GNNs. However, we found that using this method flattens the results, making them less readable.

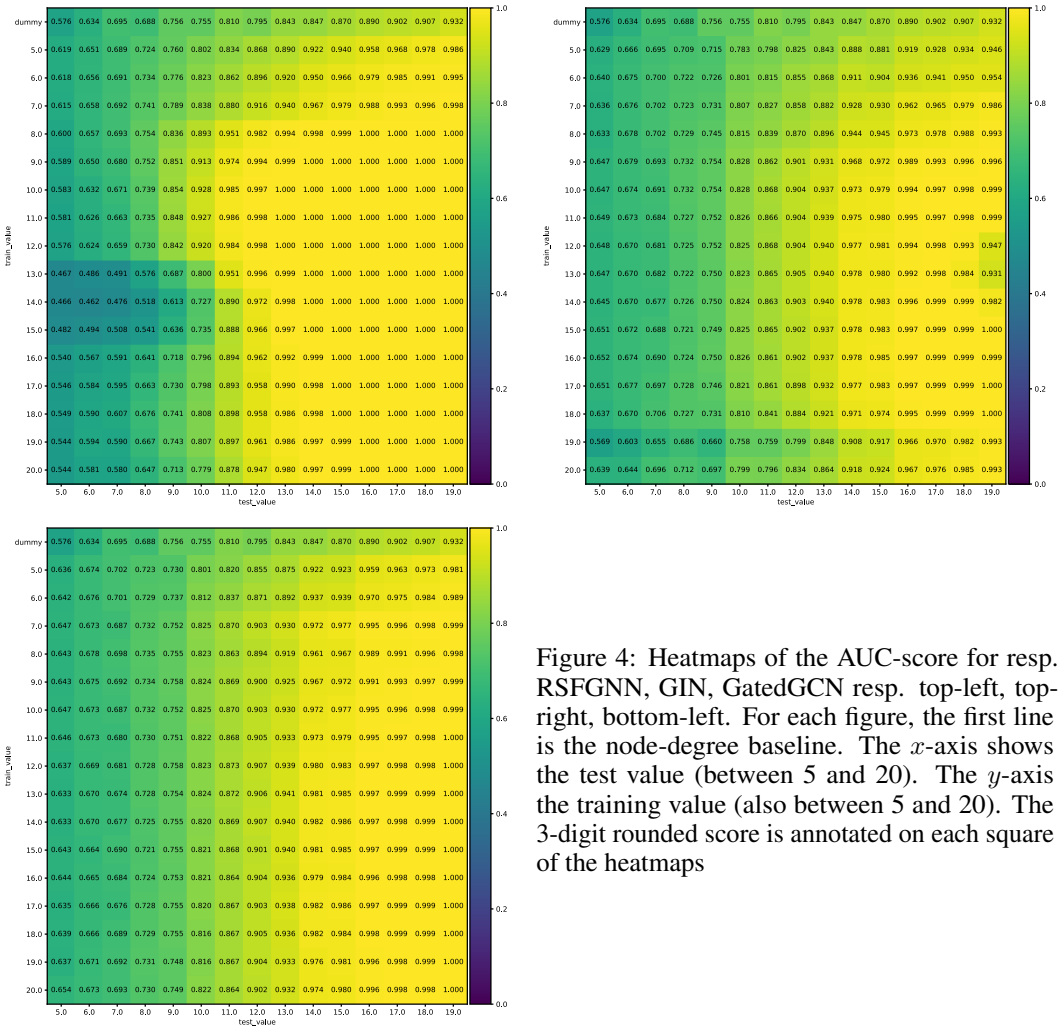

Figure 4: Heatmaps of the AUC-score for resp. RSFGNN, GIN, GatedGCN resp. top-left, top-right, bottom-left. For each figure, the first line is the node-degree baseline. The x-axis shows the test value (between 5 and 20). The y-axis the training value (also between 5 and 20). The 3-digit rounded score is annotated on each square of the heatmaps

B ADDITIONAL FIGURES

We present here more figures that we have alluded to in the main paper.

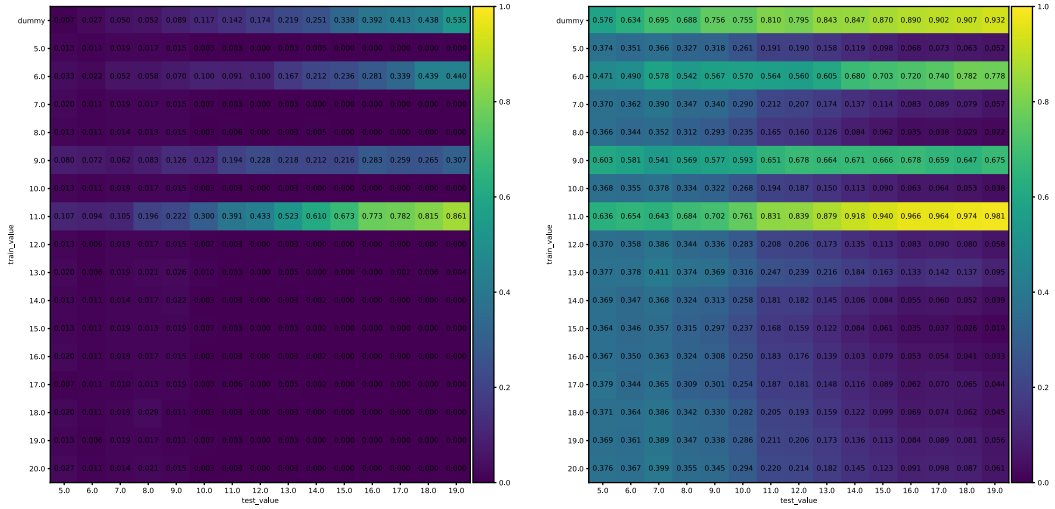

Figure 5: We show here the results for the GAT architecture (F1 on the left, AUC on the right). We haven't had the time to tweak the learning specially for this structure, thus we can see that only the models trained at clique sizes of 6, 9 and 11 have managed to learn something. We can also see that compared to the other structures, even when they have learned, they seem to perform less.