Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Solving the Maximum Clique Problem Using Neural Networks

Master's Thesis

2019
Mário Lipovský

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Solving the Maximum Clique Problem Using Neural Networks

Master's Thesis

Study programme:	Informatics
Study field:	2508 Informatics
Department:	Department of Applied Informatics
Supervisor:	Mgr. Vladimír Boža, PhD.

Bratislava, 2019
Mário Lipovský

61818992

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Mário Lipovský
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Solving the Maximum Clique Problem Using Neural Networks

Annotation: NP-hard optimization problems (like maximum clique problem) were solved using various heuristics methods in practice (e.g. branch and bound, local optimization, etc.). Our goal is to use neural networks to improve branch and bound heuristic, using neural network to guide the branch phase. We should demonstrate feasibility of the approach with practical implementation and experiments.

Supervisor:	Mgr. Vladimír Boža, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	02.05.2019
Approved:	03.05.2019

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Acknowledgement

I would like to thank my supervisor Mgr. Vladimír Boža, PhD. for providing me with an interesting topic, for his advice and for his ability to focus on positive aspects of my results, which always improved my mood.

I would also like to thank Marek Šuppa for his recommendations of PyTorch Geometric and SACRED libraries, which made the implementation process more enjoyable.

Abstrakt

V posledných rokoch boli techniky hlbokého učenia úspešne použité na riešenie kombinatorických optimalizačných úloh a pomocou myšlienok hlbokého učenia sa tiež nedávno podarilo navrhnúť nové architektúry neurónových sietí, ktoré dokážu spracúvať grafy. V našej práci spájame poznatky z týchto oblastí a pokúšame sa riešiť problém najväčšej kliky v grafe pomocou jednoduchých grafových neurónových sietí Structure2Vec a ChebNet. V prvom kroku nášho prístupu učíme tieto siete predpovedať veľkosť najväčšej kliky v okolí každého vrchola a ako trénovacie príklady pre učenie s učiteľom používame viacero tried náhodných grafov. V druhom kroku využívame predpovede grafových sietí pre rozhodovanie, ktorú vetvu navštíviť ako prvú v algoritme branch and bound. Naše výsledky ukazujú, že grafové siete dokážu detegovať kliky v malých grafoch, hoci naše experimenty zamerané na schopnosť generalizácie naznačujú, že tieto modely ešte nedokážu úplne uchopiť základný koncept klík. V našej práci tiež ukazujeme, že ak je algoritmus branch and bound usmerňovaný predpoveďami grafovej neurónovej siete, tak dokáže nájsť najväčšiu kliku efektívnejšie, než s využitím heuristickej funkcie, založenej na stupňoch vrcholov, no zatiaľ stále lepšie výsledky dosahuje s využitím heuristickej funkcie, ktorá využíva vrcholové farbenia. Okrem našich hlavných výsledkov v našej práci tiež prinášame množstvo nových postrehov, ktoré smerujú k lepšiemu využitiu grafových neurónových sietí pre riešenie kombinatorických optimalizačných úloh.

Kľúčové slová: problém maximálnej kliky, algoritmus branch and bound, hlboké učenie, grafové neurónové siete

Abstract

Inspired by recent initiatives to solve combinatorial optimization problems using deep learning and to process graphs with neural networks, in our work we try to solve the maximum clique problem using simple graph neural networks Structure2Vec and ChebNet. We use supervised learning in the first step of our approach – we train graph neural networks on various types of random graphs to predict the maximum clique size in the neighbourhood of each vertex. In the second step we then use these predictions to guide a branch and bound tree search to construct the maximum clique. Our results show that graph neural networks can learn to predict clique sizes in small graphs, although our generalization experiments suggest that these models are not able to grasp the underlying concept of cliques. We also show that when the graph network is used as a branching heuristic function of branch and bound algorithm, it can outperform degree-based heuristic, but it does not achieve the effectiveness of a more advanced heuristic based on vertex coloring. In addition to our main results, we also provide valuable insights that may improve the methodology of solving combinatorial optimization problems using graph neural networks in the future.

Keywords: maximum clique problem, branch and bound, deep learning, graph neural networks

Contents

List of Figures

List of Tables

List of Abbreviations

B&B Branch and bound

CO Combinatorial optimization

FMC Fraction-of-maxumum-clique metric

GNN Graph neural network

MCP Maximum clique problem

MIS Maximum independent set

ML Machine learning

MLP Multilayer perceptron

MSE Mean square error

MVC Minimum vertex cover

TSP Travelling salesman problem

Introduction

For its various practical and theoretical applications, the maximum clique problem (MCP) is one of the most studied NP-complete problems in computer science. In its simplest form, in MCP we search for the largest complete subgraph of an undirected graph. The most successful exact approaches to MCP use branch and bound framework, where a search tree is pruned using branching and bounding heuristic functions and the efficiency of resulting algorithms depends mainly on the quality of these functions [49].

The success of deep neural networks led researchers to apply the methods of deep learning to NP-hard combinatorial optimization problems in recent years. With this approach, the models are expected to learn to exploit structural patterns in problem instances, which can lead to automatic design of more effective heuristics. One of the main propositions of this paradigm is that in order to achieve the best results, machine learning approaches should not replace the classical algorithmic approaches, but should be used to enhance them [5, 3].

Motivated by the importance of MCP and by the fact that, to our knowledge, the techniques of deep learning were not yet directly applied to MCP, we train deep neural networks to detect large cliques in this work. We subsequently utilize these networks as branching heuristic functions in branch and bound algorithm.

Since MCP is a graph problem, we use graph neural networks (GNNs) [7], which were recently designed to process graphs in an end-to-end fashion. GNNs model invariances to vertex labels with a message passing framework, where a neuron, which is placed in a vertex of a graph, aggregates and transforms the outputs of neurons of neighbouring vertices. One of the advantages of GNNs is that a trained instance of GNN can be reused on graphs with various structures and sizes, since neurons in all vertices share a single set of parameters. Following the paradigm of deep learning, the expressive power of GNNs can be further increased by composition of multiple message passing layers.

The main goals of our work are to investigate, whether GNNs can be used to detect cliques, and whether they can improve the branching function of branch and bound. We believe that with our approach, we might improve, or at least obtain knowledge necessary to improve the capabilities of GNNs and the performance of branch and bound algorithm.

In the first chapter of this work we summarize previous efforts to solve MCP and we elaborate on motivation for using machine learning techniques for solving combinatorial optimization problems. We also describe GNNs in greater detail and we review previous deep learning approaches to problems similar to MCP.

In brief Chapter 2 we explain our motivations, main design decisions and the objectives we set to achieve in our work.

The main body of our work is contained in Chapter 3, in which we train graph neural networks to predict the size of the largest clique in the neighbourhood of each vertex and to rank vertices according to these values. Our approach uses supervised learning, so in the first step, we generate small random graphs to serve as training examples and we use an exact algorithm to compute the target values for these graphs. In our following experiments we compare the precision of GNNs and simple baselines using various types of graphs, explore the influence of architectural features on the capabilities of GNNs and we also test generalization abilities of these models.

The second part of our work, described in Chapter 4, is concerned with using predictions of these models to guide branch and bound tree search. We compare the neural heuristic function to commonly used heuristics and to the theoretically optimal branching rule.

Chapter 1

Background and Previous Work

Since deep learning methods proved useful in solving combinatorial optimization problems in recent years, we try to use these methods to solve the maximum clique problem (MCP) in our work. In this chapter we provide the background knowledge necessary for understanding our work and we also briefly review most recent efforts that are relevant to our topic.

We first introduce the maximum clique problem and we briefly summarize exact and heuristic algorithms that were used to solve it. Afterwards, we explain main features of the general machine learning paradigm and we explain, how machine learning can be useful in the context of combinatorial optimization problems. In last two sections of this chapter we describe graph neural networks and we summarize recent approaches to solve combinatorial optimization problems using machine learning and graph neural networks.

1.1 Maximum Clique Problem

We begin by formally defining graphs, cliques and the maximum clique problem. Then we outline the theoretical and practical importance of the maximum clique problem and finally, we summarize the most successful techniques for finding maximum cliques.

1.1.1 Graph Notation and Problem Definition

Definition. *Let us define a graph G as a pair (V, E), where $V = \{v_1, \ldots, v_n\}$ is a set of vertices (or nodes) and $E \subseteq \{(u, v) \mid u, v \in V\}$ is a set of edges. Two vertices $u, v \in V$ are adjacent to each other (or neighbours) if there exists an edge $(u, v) \in E$. We denote a set of neighbours of a vertex v as $N(v) = \{u \in V \mid (u, v) \in E\}$.*

Definition. *A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ if $V' \subseteq V$, $E' \subseteq E$ and $\forall (u, v) \in E' : u \in V' \wedge v \in V'$. We also define a subgraph of G induced by a vertex set $S \subseteq V$, as $G_S = (S, (S \times S) \cap E)$.*

Definition. *In a graph $G = (V, E)$, a subset of vertices $C \subseteq V$ forms a clique if each pair of clique vertices is adjacent to each other in G, $\forall u, v \in C : u \neq v \implies (u, v) \in E$. Equivalently, $C \subseteq V$ is a clique of G if G_C is a complete graph.*

In *the maximum clique problem* (MCP) we search for the largest clique in a given graph. Formally, in context of combinatorial optimization, the set of all instances of the problem is the set of all graphs, the set of feasible solutions for a graph G is the set of all cliques of G and the measure, which we try to maximize, is the clique size.

The size of the optimal solution of MCP – the size of the maximum clique of a graph G – is usually called the *clique number*, denoted as $\omega(G)$. There can be multiple cliques of size $\omega(G)$ in the graph G, and in practical applications, we are usually interested not only in the size of the clique, but also in concrete vertices of G that form the clique, i.e. in construction variant of the problem.

1.1.2 Theoretical and Practical Importance

The MCP is known to be NP-hard and NP-complete [23] and thus one could solve many other hard problems with use of clique-finding algorithms and vice versa.

For example, the maximum clique problem can be simply reduced to problems of maximum independent set (MIS) and minimum vertex cover (MVC). In MIS we search for the largest set of non-adjacent vertices and in MVC we search for the smallest set of vertices whose adjacent edges cover all edges of the graph. Let us define a complement of a graph G as $\overline{G} = (V, V \times V - (E \cup \{(v, v) \mid v \in V\}))$. If we denote the maximum clique of G as C, then the vertices of C form the optimal solution of MIS in $\overline{G}$ and vertices $V_G - C$ form the solution to MVC in $\overline{G}$.

The maximum clique problem has also relations to other combinatorial problems, such as clique partitioning, graph clustering, graph vertex coloring, max-min diversity, optimal winner determination, set packing and sum coloring as well as to numerous practical applications in bioinformatics, chemoinformatics, coding theory, economics, financial networks, location, scheduling, signal transmission analysis, social network analysis, wireless networks and telecommunications and thus the MCP became one of the most studied problems in computer science [49].

1.1.3 Exact Approaches

Most exact methods for solving the MCP are based on a back-tracking algorithm, which is further improved with a pruning technique called branch and bound. For simplicity, we will refer to algorithms that use this technique as branch and bound (B&B) algorithms.

In this section we first describe general structure of B&B algorithms and then we summarize most successful branching and bounding strategies. In the end of this section we outline other exact approaches for the MCP.

Branch And Bound Algorithm Computation of B&B algorithm can be interpreted as a tree search where nodes of the tree correspond to cliques and each edge connecting a parent node to child nodes represents an extension of the parent clique with one of candidate vertices. To make the search more efficient, bounding rule is used to prune away branches of the recursion tree that do not contain cliques larger than the largest clique found so far, and branching rule should lead to fast discovery of large cliques.

During its execution, the algorithm keeps track of the current partial clique C, the largest clique found C^* and the set of candidate vertices P. Both, the partial and the largest clique are initialized to empty sets, $C := \emptyset, C^* := \emptyset$, and all vertices are at first considered as candidates, $P := V$.

In each recursion call, branching scores $h_{br}(v) \in \mathbb{N}$ are assigned to all candidates $v \in P$ according to the *branching rule* (or *branching heuristic*) h_{br}. Candidates are then processed in order determined by their branching scores, and the vertex with the highest $h_{br}(v)$ is then selected as the first one.

Vertex v is only added to the clique if $C \cup \{v\}$, has a potential to form the new largest clique. The potential $h_{bo}(v) \in \mathbb{N}$ of each candidate $v \in P$ is determined by the *bounding rule* (or *bounding heuristic*) h_{bo}. If the rule provides an upper bound of the size of the largest clique containing v in the subgraph induced by P, then a branch of v can be pruned away if $|C| + h_{bo}(v) \leq C^*$. If the rule fails to provide this upper bound however, a part of the search tree that leads to the maximum clique might be pruned away and the algorithm might produce a sub-optimal solution.

Otherwise, if the addition of v seems promising, v is added to the partial clique creating a new clique $C' := C \cup \{v\}$. The list of candidates is then refined to $P' := P \cap N(v)$, so that it only includes neighbours of v. This refinement ensures that $P \subseteq \bigcap_{u \in C} N(u)$ and thus each candidate can be used to extend the clique.

If $|C'| > |C^*|$, the new clique is stored as the largest, $C^* := C'$, and a new recursion call is executed with arguments C' and P'. Finally, vertex v is removed from candidate set P and a new candidate is selected to extend clique C according to h_{br} and h_{bo}.

Branching and Bounding Rules A perfect branching rule would assign the highest score to vertices of the largest clique in G_P and a perfect bounding rule would precisely predict the size of the largest clique in a graph induced by $P \cap (\{v\} \cup N(v))$, or equivalently $\omega(G_{P'}) + 1$. While with use of these rules, $\omega(G)$ recursive calls would be sufficient to find the largest clique, a significant amount of computation time might be necessary to compute these rules and thus most efficient B&B algorithms have to exploit the trade-off between the quality of heuristic rules and their computational complexity.

The early B&B approaches were focused on simple and fast branching and bounding heuristic functions, while more recent approaches use more complex and more precise methods that are able to reduce the overall computation time.

Currently, the most successful branching and bounding strategies are based on the idea of vertex coloring. In the vertex coloring problem we search for the smallest k such that the vertices of a graph can be partitioned to k color classes where adjacent vertices belong to different color classes. It is easy to see that $k \geq \omega(G)$ and so vertex coloring can be used as a bounding rule and Tomita et al. [42] showed its usefulness as the branching rule as well. While finding a minimum vertex coloring is also an NP-hard problem [23], for the purposes of branching and bounding, fast heuristic coloring procedures can be used instead of exact coloring algorithms. We explain these coloring-based heuristics in greater detail in Section 4.1.2.

The coloring approach was incrementally improved by initial ordering of vertices according to their degrees [41, 28], improving the coloring subroutine to achieve lower number of colors [43], using bit-parallel operations [36], solving max-SAT instances to achieve tighter bounds [30] and by using a heuristic local search algorithm to find a large clique, which is used as the initial lower bound C^* [2].

Other Exact Approaches to MCP Other techniques were also explored, but they were surpassed by the aforementioned B&B approaches for now. These include computing bounds with iterative deepening [34], using only a single initial whole-graph coloring for bounding [29], computing multiple colorings and directly removing unpromising and including promising candidates [14].

The comparison of these algorithms, as well as their brief but more detailed explanations, can be found in a clique review paper by Wu et al. [49].

1.1.4 Heuristic Approaches

Although approximation algorithms are also commonly used to solve hard problems, the maximum clique problem is hard to approximate. Since for any $\varepsilon > 0$ no approximation algorithm within a factor $n^{1-\varepsilon}$ exists [19] (unless $P = NP$), clique approximation algorithms are not useful in practice. Non-exact approaches for MCP are thus limited to heuristic algorithms with no strong guarantees.

Most successful heuristic approaches use the framework of the local search, where a population of cliques is modified with add, swap and drop operators on vertices to search the space of all cliques or of all cliques of a given size.

Other successful approaches are based on a sequential greedy construction of a clique using neighbourhood statistics or dynamically computed weights from multiple restarts to select the next vertex to add to the clique.

An interesting observation made by Wu et al. [49] is that there is no effective recombination (or crossover) operator known for MCP and thus evolutionary algorithms are not competitive to simpler local search approaches.

1.2 Introduction to Machine Learning

After introducing the MCP and previous approaches that were used to solve it, we can shift our attention to machine learning (ML) methods we intend to use in our work.

Before moving towards more complex graph neural networks, we start with a brief introduction of the fundamental concepts and terms necessary for understanding the machine learning approach, neural networks and specifics of deep learning. With clear understanding of ML paradigm we can then present the motivations to use ML methods for combinatorial optimization problems such as MCP in Section 1.2.3.

The core idea of solving a problem with ML is to create a program that can be automatically taught to perform a task, e.g. by observing examples, instead of programming a set of hand-crafted decision rules.

Model In machine learning, the program that is trained is usually called a *model*, and to solve a problem we first have to specify, what the model can be taught. From a mathematical point of view, the process of *training* a model can be defined as a selection of one function, a hypothesis, from a specified set of hypotheses. To enable a mathematical analysis of a model, the hypotheses set is most often parametrized with parameters θ. Training a model is then simplified to selection of the best parameters $\hat{\theta}$ which define the hypothesis $h_{\hat{\theta}}$.

Learning Paradigms When an architecture of the model (a set of hypotheses) is selected, the specific abilities of the model (the final hypothesis) depend mostly on the learning process and on the examples which are provided for training. There are three main paradigms of learning:

- Unsupervised: examples consist only of inputs and the model is expected to learn compact representation and/or distribution of the inputs.
- Supervised: examples consist of pairs (input, desired output) and the model is usually trained to predict the output labels based on inputs.
- Reinforcement: the model is in a role of an agent and is expected to learn a behaviour that leads to a specific goal – the model manipulates an environment

by deciding which actions to take and tries to maximize the received reward signal.

Loss Function To define, which hypothesis is the best for solving the task, a *loss function* (or an *error function*) $l(\theta, t)$ is used, where t denotes one training example. For example, in case of supervised learning, $t = (x, y)$ might be a pair of input x and desired output y and the quadratic loss $l(\theta, (x, y)) = (h_\theta(x) - y)^2$ might be used. By minimizing the loss, we might be able to find a hypothesis for which $h_\theta(x) = y$ holds for all training examples.

Training With the loss function defined, training a model is most often a non-convex continuous optimization task that has to be optimized with metaheuristics, such as gradient descent. In gradient descent we iteratively update the parameters in the direction of the most steep descent of the loss – in the direction opposite to the gradient: $\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_\theta l(\theta)$.

Generalization To create a program that would reproduce desired outputs for observed inputs, the simplest solution would be to implement a lookup table. We would store all the input/output pairs in memory and then retrieve the memorized output for a given input at inference. However, in machine learning we are trying to train models that can generalize – solve the task for all input instances and, most importantly, for the unobserved ones.

To make this possible, we have to assume that there are correlations between inputs and outputs and that these patterns don't change over time. To formalize this point of view, we can assume that all examples come from a joint probability distribution P. To achieve the best general solution, we would have to minimize $\mathbb{E}_{t \sim P}[l(\theta, t)]$, but since we only have a limited number of examples and a limited training time, we have to approximate the expected value with $\frac{1}{|T|}\sum_{i=1}^{T} l(\theta, t_i)$.

To measure the generalization ability of a model, we usually split the available examples into a training subset and a testing subset. The model is then trained with training examples and the loss is eventually evaluated on previously unseen testing examples.

Precision Metrics In addition to the goal of achieving generalization abilities of models, there is one more main aspect of ML that differentiates it from continuous optimization.

Architecture of models and the nature of the training algorithm often limits the possible space of loss functions. For example, to use gradient descent, loss function and hypotheses have to be differentiable (or differentiable in parts) functions, but in case of classification tasks, the desired outputs y might be from set $\{0, 1\}$ and the goal might be the maximization of $\frac{1}{|T|} \sum_{i=1}^{T} [h_\theta(x) = y]$ where $[a = b]$ is 1 if $a = b$ and 0 otherwise.

To make the ML approach useful also for tasks where the goal is not easily expressible in a required form, a subsidiary loss function, e.g. quadratic loss, is used to set the parameters θ, but the final model variant might be selected from a set of trained models according to the *precision metric* that is more relevant to the final goal.

1.2.1 Neural Networks

Neural networks provide a modular framework that can be used to represent a broad class of functions (or hypotheses) and are often used to define ML models.

The simplest neural network, a linear layer, consists of a matrix multiplication of the input vector with a weight matrix and an addition of a bias vector. This network is able to learn to perform an affine transformation $f(x) = W \cdot x + b$.

Since a composition of affine transformations only yields an affine transformation, to expand the space of functions that can be represented with a neural network, point-wise non-linear transformations, such as rectification $\text{relu}(x) = \max(0, x)$, can be used. More complex neural networks are then constructed as a composition of simpler functions.

A simple example of a neural network is multilayer perceptron (MLP) – a composition of multiple linear layers and non-linearities. If we denote a composition of the linear layer and the non-linearity as $h_i(x) = \text{relu}(f_i(x))$, MLP with three layers can be expressed as $f_{MLP}(x) = h_3(h_2(h_1(x)))$. When all of the composed functions are differentiable, the neural network can be trained with gradient-based methods, such as gradient descent.

1.2.2 Deep Learning

Most machine learning breakthroughs in recent years were achieved with the use of deep neural networks that are able to represent more complex functions.

The core idea of deep learning is that apart from the end task (e.g. classification or regression), the model also learns how to pre-process the input data in lower layers and composes these representations into more complex concepts that are finally utilized for the task executed by the final layer [17, Chapter 1]. This end-to-end approach contrasts with the classical ML where the input examples were at first processed into sets of hand-crafted features and the trained model was only used to solve the task using these features.

While almost any real function can be potentially expressed with a deep enough neural network, networks with too many parameters are very hard to train and are also prone to over-fitting – memorizing the training examples instead of generalizing. The success of deep learning thus does not only come from the increased number of layers, but also from architectures that incorporate assumptions about application domains.

For example, translation invariance in images is exploited in convolutional neural networks. Since the same object can appear in different locations of an image, one filter (implemented with a single neuron) can be applied to all locations of the image to detect the object. As the same set of parameters is reused in each image location, this approach significantly decreases the number of the model parameters and contributes to its efficiency, simpler training process and increased generalization ability.

These architectural features of convolution and recurrent neural networks, jointly dubbed as inductive biases [3], recently inspired researchers to develop new architectures that can exploit the structure of graphs and process graphs as inputs.

1.2.3 Motivation for Using ML for Combinatorial Optimization

Machine learning approaches proved most useful for problems where the variety of inputs exceeds human capabilities to design rules to process them effectively. Therefore, the fact that NP-hard CO problems are currently solved with a wide range of heuristics presents a perfect opportunity for the use of the ML techniques.

Automatic Acquisition of Expert Knowledge Approximation and heuristic algorithms exploit structural patterns of a problem instance. To produce an algorithm in a conventional way, however, these patterns have to be at first noticed by a human, then a way of exploiting this knowledge must be discovered and finally, the algorithm has to be implemented.

As the tasks of pattern recognition in text, images and sound were most successfully solved by deep learning approaches, we hope that these approaches will also be able to find patterns in combinatorial structures, learn their representations and combine representations of simpler structures to represent more complex ones. This approach might be much faster and might possibly lead to discovery of more relevant patterns than those that were acquired with human expert knowledge.

Exploiting the Domain NP-hard problems are hard to solve in general case, but in particular instance families it might be easier to find the optimal solution, and heuristic approaches can exploit these features of instances. While this approach can improve a performance of a heuristic on a particular type of instances, it also leads to inconsistent performance on different types of instances and thus specific heuristics might be necessary for solving specific instances.

While simpler machine learning models can be trained to select the best heuristic for each instance, deep models can be trained on examples that come directly from the application domain and learn to distinguish typical patterns. As a result, these models might outperform domain-independent methods.

A Single General Approach Although multiple combinatorial problems share common elements, e.g.: the input might be a graph or a set and the solution might be a subgraph or a subset, conventional algorithms and heuristics share the same drawback that even a slight change in the task setting demands a change in the algorithm.

In contrast with classical algorithms, a single machine learning algorithm might be easily reusable for a wide spectrum of tasks when trained on different training sets with different values of hyper-parameters. Moreover, with reinforcement learning a model can learn to decide, which subroutines to execute in specific states of the optimization process and thus it can lead to the design new algorithms.

1.3 Graph Neural Networks

Since MCP is a graph problem, a question of "how to represent a graph as an input for a neural network" arises. Moreover, as we would like to solve MCP for graphs that were not observed before, this representation has to be independent of the vertex labels. Graph neural networks (GNNs) solve both of these problems.

To highlight the abilities of GNNs, we first briefly mention previous ML approaches that were used to process graphs in this section. Afterwards, we introduce GNNs by describing their common message-passing architecture and we summarize the ideas that led to their design. In the end of this section we then present a few recent theoretical results about the capabilities of GNNs.

With the understanding of deep learning and GNNs, we will be finally ready to review previous methods that were used to solve CO problems in graphs.

1.3.1 Processing Graphs With ML

There are many kinds of real-world problems that can be represented as graphs and so a variety of methods was developed to exploit the graph structure of inputs with machine learning models.

In a classical machine learning setting, the first model usually embeds vertices, edges or whole graphs into a common space $\mathbb{R}^k$ so that their embeddings retain either their proximity or structural similarity. These models are usually based on factorization of adjacency matrix, sampled random walks or on kernel methods [9]. The second model is then usually used to process these embeddings into final predictions.

While these methods might be effective for some problems, methods based on adjacency matrix factorization usually only focus on retaining the distances between vertices and do not capture more complex features of the graph structure.

Methods based on random walks use vertex labels to distinguish vertices from each other and so the final embeddings are specific for a single input graph and the learned knowledge is not transferrable to other graphs. Although random walk methods can compute new embeddings for new graphs, they cannot guarantee that two vertices with a structurally similar surroundings will share similar embeddings if these vertices come from different graphs. Therefore, only models that process all vertices of a graph all at once (e.g. clustering methods or possibly recurrent neural networks) might be able to use the information encoded in these embeddings.

Graph kernel methods are usually used for whole-graph embeddings since they represent a graph as a set of particular subgraphs contained in it. Although it might be possible to represent surroundings of each vertex with a set of subgraphs, only substructures relevant to MCP for a vertex v, would be subgraphs of the immediate neighbourhood of v – the subgraphs of $G_{\{v\}\cup N(v)}$.

1.3.2 Architecture of GNNs: Message-Passing Framework

Graph neural networks use both the depth (the end-to-end approach and the large number of layers) and inductive biases (an invariance to vertex labels and an invariance to the order of neighbouring vertices) to learn representations of graph substructures that are relevant for the final task. Since they proved successful in various fields, such as chemoinformatics [16, 10], 3D shape analysis [7] and combinatorial optimization [11, 31], they currently gain more and more interest.

Currently there are many different architectures of GNNs, but they mostly share the same message-passing structure [16, 3, 15].

Graph Convolutional Layers In its simplest form, a GNN usually consists of multiple graph convolutional layers, where i-th layer executes i-th step of a message passing algorithm and computes new hidden embeddings $h^{(i)}$ using the outputs of the previous layer, $h^{(i-1)}$. A computation preformed by a single layer can be described in three steps. In the first step, *messages* are computed for all pairs of adjacent vertices. In the second step, the messages of neighbourhood vertices are *aggregated* for each vertex, and finally, a new embedding is computed, or *updated*, using the previous state of the vertex $h^{(i-1)}$ and the aggregate of messages.

$$h_v^{(i)} := \text{update}\left(h_v^{(i-1)}, \text{aggregate}_{u\in N(v)}\left(\text{message}\left(h_v^{(i-1)}, h_u^{(i-1)}\right)\right)\right)$$

While messages and updates can be computed with common neural networks, permutation invariant function, such as summation, is usually used for aggregation. Hence the invariance to the order of neighbouring vertices is ensured.

Initial embeddings $h^{(0)}$ usually contain input features describing the properties of vertices, e.g. personal information for people in a social network or properties of atoms in a molecule. Since in our task we will only process simple graphs with no additional information, all vertices will have $h^{(0)}$ set to the same constant value, which will ensure the invariance to node labels.

Notice that the same set of parameters is shared across all vertices in a single layer, and thus the number of trainable parameters of GNN is independent of the graph size. Moreover, a single instance of GNN can be deployed on any graph by copying its topology.

Usage of GNNs We can formally view a graph neural network g_θ as a function parameterized with θ that assigns a k-dimensional real embedding vector $h_v \in \mathbb{R}^k$ to each vertex $v \in V$ of a graph G:

$$g_\theta : G \mapsto (h_{v_1}, \ldots, h_{v_{|V|}})$$

After multiple iterations of message-passing, the embeddings can be either used as inputs for another neural network $f : h_{v_i} \mapsto \mathbb{R}$ in case of regression or classification task on vertices. In tasks where a single value is predicted for the whole graph, vertex embeddings are usually first aggregated with a *readout* function, e.g. $\text{readout}(h_{v_1}, \ldots, h_{v_{|V|}}) = \sum_{i=1}^{|V|} h_{v_i}$, and the readout value is then passed to the output network f.

To condition learned vertex or graph representations on the final task, the parameters of convolution layers of g are optimized jointly with the parameters of the output network f. This ability enables GNNs to process graphs end-to-end as opposed to a two-step approach mentioned in Section 1.3.1, and to optimize the embeddings to capture the structural information that is most relevant for the final task.

Extensions Extensions of the basic architecture of GNNs include computation of additional embeddings for each edge, sharing parameters across convolutional layers, additional *pooling* vertices that aggregate the information from all $h_v^{(i-1)}$ during message-passing steps and provide it for computation of $h_v^{(i)}$, use of recurrent neural networks for update computation, contractions of the graph in higher layers and more. The most general formalization of GNNs was provided by Battaglia et al. [3].

1.3.3 The Inception of GNNs

The concept of GNNs was re-invented multiple times in the recent years [38, 8, 10]. We consider the fact that each of these approaches introduces GNNs as a generalization of a different class of models as fascinating, since it shows the possible directions for the future applications, analysis techniques and theoretical research.

Recursive Networks and Random Walks Graph neural networks were first mentioned by Scarselli et al. [38] as a generalization of recursive neural networks operating on directed acyclic graphs, and of random walks in Markov processes. In this approach, a message-passing step using a single set of parameters was repeated until convergence. Since this work was published before the start of the deep learning revolution, according to a recent review on GNNs [7], this work remained mostly unnoticed.

Graph Convolution, Spectral Analysis and Fourier Transform The first reincarnation of GNNs in the era of deep learning came from Bruna et al. [8] where the authors were inspired by the success of convolutional image processing architectures. In their approach, the authors first defined a convolution operation in non-euclidean spaces (graphs and manifolds) with use of the spectral decomposition of the graph's laplacian matrix. Since this approach was computationally demanding, a modification that avoided the costly eigenvector computation was proposed by Defferard et al. [12] and with this modification, the whole approach became describable in the message-passing GNN framework.

The computation of one layer of one such network, *ChebNet* [12], can be described by matrix operations

$$H' = \sum_{k=0}^{K-1} Z_k \cdot W_k$$

$$Z_0 = H \qquad Z_1 = L \cdot H \qquad Z_k = 2 \cdot L \cdot Z_{k-1} - Z_{k-2}$$

where H, H' are the matrices of old resp. new hidden states (the embeddings of vertices are represented as row vectors), Z is a Chebyshev polynomial, W_k is the weight matrix for k-th summand of the polynomial and L is a normalized laplacian matrix $L = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ where A is the adjacency matrix of the graph and D is a diagonal matrix of vertex degrees.

Since the multiplication of the adjacency matrix A with H computes the sum of embeddings of neighbouring vertices for each vertex, the computation of the ChebNet's layer can be equivalently expressed in the message-passing framework for $K = 2$ (and with use of multiple graph convolutional layers also for higher K) as:

$$h_v^{(i)} := h_v^{(i-1)} W_0^{(i)} + \sum_{u \in N(v)} \frac{1}{\sqrt{\deg(u)\deg(v)}} h_u^{(i-1)} W_1^{(i)}$$

This approach shows the connections of GNNs to convolutional networks and spectral graph theory. We summarized the main ideas of this approach in Appendix A – Spectral GNNs, and its long version can be found in a review paper on geometric learning [7].

Graphical Models A simple class of GNNs based on graphical models, *Structure2Vec* (S2V), was proposed by Dai et al. [10]. In their approach, the authors enhance message-passing algorithms in Markov network model by replacing the costly operations on probability distributions with neural networks. The authors use the assumption that any probability distribution P of a random variable X can be injectively embedded into a point μ_X of Euclidean space of large enough dimension using a kernel function ϕ to transform realizations x of X, $x \mapsto \phi(x) \in \mathbb{R}^k$:

$$\mu_X = \mathbb{E}_{X\sim P}\left[\phi(X)\right] = \int_{\mathcal{X}} \phi(x)p(x)dx$$

With distributions embedded in $\mathbb{R}^k$, the authors also embed the operations on distributions into this space. As a result, neural networks can be used to learn and perform analogies of these operations. The computation of one step of embedded mean-field inference in Markov network can be then expressed as the following message-passing step:

$$h_v^{(i)} := \operatorname{relu}\left(h_v^{(0)}W_0 + \sum_{u\in N(v)} h_u^{(i-1)}W_1\right)$$

Notice that in this approach, an initial vertex tag $h_v^{(0)}$ is used repeatedly and that the same set of parameters W_0, W_1 is used across all layers, since the belief propagation step was usually iterated until convergence in graphical models.

Although this approach leads to an algorithm that is more efficient than inference in graphical models, it might be interesting to analyze, whether embedded operations obtain some properties of the original operations of probabilistic graphical models in the process of learning.

1.3.4 Theoretical Limitations of GNNs

A recent line of work showed that capabilities of GNNs are strongly related to Weisfeiler-Lehman algorithm for graph isomorphism [47, 52, 32].

In this algorithm, the same starting tag $h^{(0)} \in \mathbb{N}$ is assigned to all vertices. New tags are then computed for all vertices with an injective function (usually implemented with a hash table) $T : \mathbb{N} \times 2^{\mathbb{N}} \to \mathbb{N}$ in each consecutive iteration. If we denote all neighbours of a vertex v as $u_1, \dots, u_m$, then

$$h_v^{(i)} := T\left(h_v^{(i-1)}, \left\{h_{u_1}^{(i-1)}, \dots, h_{u_m}^{(i-1)}\right\}\right)$$

After each iteration, the sequences of tags of all graph vertices $\left(h_{v_1}^{(i)}, \dots, h_{v_n}^{(i)}\right)$ are compared for two graphs and if these sequences do not match, the graphs are found to be non-isomorphic. Otherwise the iterations continue until the convergence of tags.

In the recent work it was shown that message-passing GNNs have the same ability to distinguish non-isomorphic graphs (and the local neighbourhoods of vertices) as the Weisfeiler-Lehman method, if summation is used as aggregation function and if the update function is complex enough to provide injectivity [52]. Moreover if these conditions are not met, for example, if mean or max functions are used for aggregation, the ability of GNN to distinguish substructures deteriorates.

Based on these observations, a simple architecture

$$h_v^{(i)} := \mathrm{MLP}^{(i)} \left((1+\varepsilon) \cdot h_v^{(i-1)} + \sum_{u \in N(v)} h_u^{(i-1)} \right)$$

was proposed by Xu et al. [52] and was empirically shown to achieve the performance of the state-of-the-art GNNs as well as kernel methods based on Weisfeiler-Lehman-subtrees [39].

For Weisfeiler-Lehman method it is well known that it cannot distinguish non-isomorphic k-regular graphs of the same size and k [13] and so GNNs also cannot distinguish regular substructures of a graph. These limitations motivate the research to find stronger variants of graph-processing networks, e.g. networks that would be able to express more general k-Weisfeiler-Lehman method [32].

1.4 Machine Learning & Combinatorial Optimization

There are many possibilities, how to approach CO problems with ML. To provide a wider context for our work, we start this section with a brief cross-section through various ML methods that were used to solve CO problems. We move from the simplest ML methods to the most recent, most complex and most relevant methods, which use graph neural networks. We finish this section by summarizing main features of current ML/CO methodologies and the challenges that ML has yet to face in the field of CO.

1.4.1 ML Approaches to CO

Although there have been attempts to use biologically motivated neural networks, such as Hopfield networks, to solve the travelling salesman problem (TSP) [4, previous work] and the MCP [20, 46]. For TSP these approaches did not turn out competitive [37, 4], however, and for MCP only a limited comparison to older approaches was carried out. Moreover, since Hopfield networks do not learn any transferrable knowledge, they cannot be considered as machine learning methods.

Classical Machine Learning The use of classical machine learning was successful in the field of mixed integer programming, where relatively simple models, logistic regression and support vector machines, were used to predict the next variable to set [24] or a set of heuristics to run [25] in each node of a branch and bound algorithm.

A different approach was used to enhance a mixed integer quadratic programming solver, where a machine learning model is used in the beginning of the computation to determine, whether the linearization of the instance will lead to faster computation [6].

The First Use of Deep Learning To our knowledge, deep learning techniques were first used for combinatorial optimization in 2015 by Vinyals et al. [45].

The authors solve the metric TSP, the convex hull problem and the Delaunay triangulation problem using a recurrent sequence-to-sequence architecture [40] which enables the model to process inputs (a sequence of 2D points) of various sizes. As their model, a *pointer-network*, is also enhanced with an attention mechanism [1] it can produce elements of the input set as its outputs and thus solve the problem in an end-to-end fashion – directly generate points in the travelling salesman route. The final solutions are then obtained using a greedy inference algorithm.

According to their results, their model can generalize from training examples of size 50 to testing examples of size 500 for the convex hull problem, but for the TSP, the generalization from 20-vertex training graphs only holds up to 30-vertex testing graphs.

Pointer-networks were later used to solve the metric TSP with reinforcement learning and with use of more sophisticated inference procedures [4]. In this setting the authors were able to produce close-to-optimal solutions even for graphs with 100 vertices.

Solving Graph CO Problems With GNNs While the previous approaches did not exploit the graph structure of the problems, a general architecture, S2V-DQN, based on the Structure2Vec network (see Section 1.3.3), was proposed by Dai et al. to solve all combinatorial optimization problems that involve graphs [11].

This approach operates with a Q-learning framework, where a GNN is repeatedly used to compute Q-values for candidate vertices and the final solution is constructed with greedy inference guided with Q-values. To avoid the construction of subgraphs induced by the current candidate set, the vertices in the candidate set are marked with initial tags $h^{(0)} = \vec{1}$ and the vertices outside of current candidates are marked with $h^{(0)} = \vec{0}$. An illustration of the S2V-DQN framework can be seen in Figure 1.1.

When S2V-DQN is compared to the pointer-network trained with reinforcement learning [4] that is further enhanced to operate on simple vertex embeddings to incorporate graph structural information, S2V-DQN achieves similar performance on the TSP, but significantly outperforms pointer-networks on minimum vertex cover and

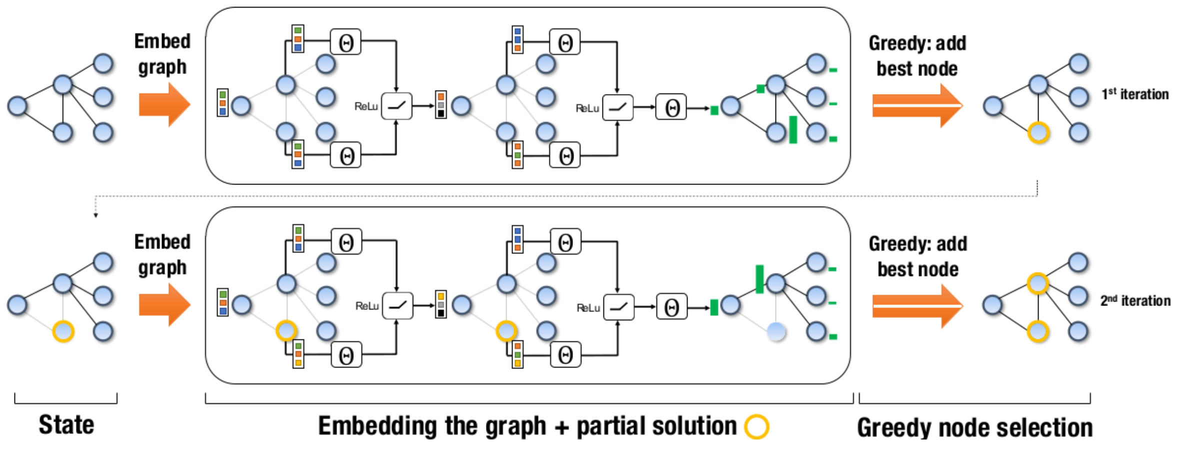

Figure 1.1: An illustration of the computation of S2V-DQN framework applied to the minimum vertex cover problem. Reprinted from the original paper [11]. The triplets of colorful squares represent vertex embeddings computed with S2V network and the green bars represent the Q-values.

maximum cut problems. Furthermore, S2V-DQN can generalize from training instances of size 100 to testing instances of size 1000.

An even more recent work by Li et al. solves the problem of the maximum independent set using a more complex ChebNet [12], but a simpler, supervised, learning approach [31]. The network is trained on 40 000 SAT instances converted to MIS instances with 1 200 vertices.

Since the targets only include binary information (whether a vertex is in the optimal solution or not), and there are many possible optimal solutions for each instance, the predictions of the GNN often contain multiple averaged solutions. To overcome this problem, the authors designed a novel inference framework which computes multiple probability maps (outputs of multiple GNNs) in each step and maintains a queue of partial solutions that are expanded with a strategy similar to breadth-first search, which explores more potential solutions. This search strategy is more exploratory than the previously used greedy inference.

To achieve even better results, the authors use simple heuristics to reduce the subgraph induced by candidate vertices before passing it to the GNN and they also use a local search heuristic to improve final solutions. An illustration of the steps used in this approach can be seen in Figure 1.2.

This approach is also tested on the instances of SAT, MVC and MCP using reductions to MIS and can outperform S2V-DQN. This approach solves all tested instances of SAT and also finds MIS of the same size as a state-of-the-art MIS solver in real-world graphs. When this approach is used for maximum clique problem, however, it only solves 62, 5% of the testing instances and the authors expressed a need for a

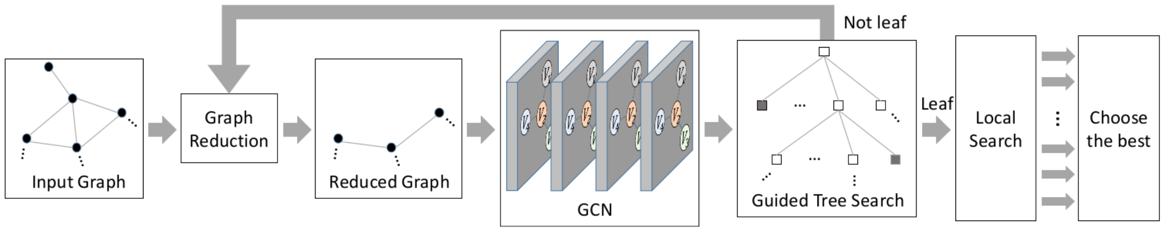

Figure 1.2: An illustration of the computation of the framework by Li et al. Reprinted from the original paper [31]. A graph is repeatedly reduced to only contain promising vertices and multiple probability maps, which determine the probability that a vertex will be expanded, are computed using GNNs. In the least step, solutions are post-processed with a local search heuristic.

specialized approach to MCP.

Currently, there exist many different approaches that try to solve combinatorial optimization problems by combining traditional algorithms with machine learning methods and their systematic overview can be found in review paper by Bengio et al. [5].

1.4.2 Design, Methodology and Challenges of ML for CO

As we have seen in the previous section, in addition to the choice of ML model, main differences between the existing ML approaches to CO problems are in the choice of the learning paradigm and in the algorithmic structure used to convert continuous predictions of the model to discrete feasible solutions. Furthermore, in the evaluation phases of the previous studies, a special attention is paid to the ability of the models to generalize on the instances of different size and structure.

In this section we provide a mixture of our own observations and observations made by Bengio et al. [5] relevant to these topics.

Learning Paradigm The apparent difference of supervised learning compared to reinforcement learning is in the need for labeled training examples. In case of CO, a training example consists of a problem instance and of a target in form of a single optimal solution, a set of multiple solutions or an annotation of the instance that holds the information about optimal solutions.

Since large instances of NP-hard problems require considerable computation time to be solved with exact approaches, the researcher either has to invest to computation power sufficient to solve large instances, or is limited to use exact solutions of smaller instances and sub-optimal heuristic solutions of larger instances as targets.

Bengio et al. [5] argue that models trained with supervised learning using sub-optimal solutions only learn to mimic behaviours of heuristics and should be only used if they lead to faster implementations of these heuristics. On the other hand, reinforcement learning has a potential to lead to the design new, more effective algorithms. We think that in their review, it is not highlighted enough that the models trained on optimal solutions in supervised fashion might as well outperform classical approaches [31] and also lead to new algorithms, when the behaviour of the search algorithm that encapsulates the model is studied.

We agree with the authors that it might be a good idea to combine supervised and reinforcement learning by first pre-training the model with supervised learning and then improve the decision policy based on experiences of the model.

The main challenge for supervised approach, as well as for reinforcement approach, is the selection of training instances. While random families of instances might be generated much easier in CO than in the fields of image, text and sound processing, it might be hard to determine which families to use for training and evaluation of models. Real-world instances of CO problems should be collected on a larger scale in the future to exploit the full potential of ML approaches, following the example of other areas where ML proved to be successful [5].

Algorithmic Structure The simplest way how to utilize a ML model for CO is to use it to make a single decision in the beginning of an algorithm, e.g. to select a heuristic that should be run on a specific instance [6].

On the other hand, the most ambitious approaches try to directly predict optimal solutions, e.g. as a sequence of points outputted by a recurrent pointer-network [45], but since current approaches cannot guarantee feasibility of the outputted solutions, special algorithms must be used to construct feasible solutions. For example, in the case of pointer-networks, the attention mechanism of the network might assign the highest output probability to a point that was already outputted before, but to ensure the feasibility of the solution, only the points that were not previously outputted are used as possible outputs in each inference step. The design of architectures that can produce feasible solutions remains as a challenge for the future research.

The recent work shows that in order to achieve the best results, machine learning approaches should not replace the classical algorithmic approaches, but should be used to enhance them [5, 3]. One example of this approach was an improvement of the pointer-networks made by Bello el al. [4], where a beam search was used to construct multiple solutions based on the predictions of the network.

Current architectures can produce outputs in a form of a continuous distribution for elements of an input instance and these predictions can be utilized in different algorithmic structures. While we have seen examples of usage of ML models as guiding heuristics in greedy search, beam search, randomized breadth-first-search and branch and bound search, there is still a lot of space left to come up with novel usages of ML models in other settings.

Generalization to Size and Structure Generalization ability of models is interesting from both practical and theoretical point of view. For practical applications it is important to know the limitations of the model's performance on instances of different sizes and with different properties than those seen in training. On the other hand, one of the theoretical goals of artificial intelligence is to build models that are able to learn to understand the underlying principles of problems and generalization experiments present a method that tests this ability.

We think that the ability of models to distinguish all substructures that are setting apart optimal solutions from sub-optimal is necessary for grasping the principles of the problem. Therefore, as suggested by limitations described in Section 1.3.4, it would be unrealistic to expect current models to learn to understand the cores of the problems. This point of view might explain some of the negative results of generalization experiments.

Even if a model is able to detect some important structures of input instances, another challenge is to select training instances that contain diverse substructures. Training a model on instances that are too similar to each other might lead to overfitting to training data and so a slight variation in input might result in large prediction error.

Finally, ML models for CO are usually parametrized with a constant number of parameters to enable them to process inputs of various sizes. If these parameters only capture the statistical information about instances, however, ML model might fail to produce meaningful predictions for larger instances or instances of different types.

Since only training larger models on larger instances might lead to issues with complexity and their ability to learn, perhaps more attention should be paid to the development of models that are, at least in theory, able to understand the core of the problem they are trying to solve.

Chapter 2

Our Solution

In our work we apply the techniques of deep learning to the maximum clique problem.

To our knowledge, there were no previous efforts to solve the MCP in the context of deep learning at the start of our work. As we mentioned in Section 1.4.1, Dai et al. [11] tried to solve a related problem of MVC but the authors did not evaluate their approach on MCP and when Li et al. [31] evaluated their model on MCP instances reduced to MIS, it achieved considerably worse performance than when evaluated on instances of SAT, MIS and MVC problems.

We decided to approach the MCP directly, without reductions to other problems.

As branch and bound framework is currently the most successful approach to solve MCP and a significant part of its success is owed to the quality of its branching and bounding heuristic functions, our initial motivation was to create a ML model that would provide one or both of these functions.

The whole MCP and also the more specific problem of designing heuristic functions might be approached with a variety of ML methods. In order to limit the scope of our work to a manageable size, we made a few design decisions and we selected particular goals we hoped to achieve. We present these decisions in two sections of this brief chapter.

2.1 Initial Design Decisions

From B&B to Task Definition As we already mentioned above, our first idea was to substitute heuristic functions in B&B with trained models in hope to improve the exact algorithm for finding the maximum clique.

In B&B, the branching function should order the vertices of a graph (or a subgraph) from the most promising to the least promising – by the size of the largest clique in which they are contained and the bounding function should provide an upper bound of the size of this clique (1.1.3).

Since currently ML models cannot provide guarantees for their outputs, the goal of creating a learned feasible bounding function seems unrealistic. Therefore, the learned branching function might be used either in combination with a classical bounding function and produce an exact algorithm, or a heuristic might be produced by using a learned bounding function. If we decide to compare our approach with heuristics however, other algorithmic structures, such as breadth first search used by Li et al. [31], might find larger cliques faster due to their more exploratory nature.

We resolved this issue by separating the training phase of models from their incorporation in B&B. With this approach, whe can first focus on the ability of GNNs to distinguish vertices of large cliques from vertices of small cliques and later, we can use predictions of these models to guide the B&B algorithm, or possibly other algorithms.

Definition. *Let $G = (V, E)$ be a graph. We define the clique number of vertex $v \in V$, denoted as $\omega(v)$, as the size of the largest clique in the subgraph induced by vertices $\{v\} \cup N(v)$ in G.*

With $\omega(v)$ defined, we can now also define tasks that should be solved with neural networks. Analogically to the goal of the optimal bounding rule, we will train selected networks to predict values of $\omega(v)$ for each vertex, but since the precision of clique size predictions might be not as important as the ability to distinguish vertices of smaller $\omega(v)$ from vertices of larger $\omega(v)$, we will focus on a task of ordering vertices according to their clique numbers.

Learning Paradigm: Supervised Motivated by its relatively low requirements on computational power and supported by the results of Li et al. [31], we decided to train our models with supervised learning and with use of small, optimally solved instances of randomly generated graphs.

In our approach, we label the vertices of training graphs with exact values of $\omega(v)$. Since with knowledge of these labels, the maximum clique could be constructed with only $\omega(G)$ recursive calls of B&B (see Section 1.1.3), we can say that these labels carry information about the maximum clique. Furthermore, these labels also carry information about all optimal solutions and also about smaller cliques. Compared to the approach of Li et al. [31], where binary labels only denoted one optimal solution, our labels might have potential to provide the trained model with more information and possibly avoid the problem of predicting multiple solutions simultaneously.

Model and Input Representation: GNNs As neural networks expect real vectors in inputs, one of the first challenges we faced, was the question of how to represent vertices of a graph as real vectors.

After reviewing multiple approaches (see Section 1.3.1) we decided to use graph neural networks, since they seem to be able to capture more structural information than other approaches and their invariance to vertex names makes their knowledge transferrable to new graphs. Finally, their ability to be jointly trained with the network that performs the final task might bring the benefits of deep learning, which we described in Section 1.2.2.

Although many architectures of GNNs were proposed, we decide to use relatively simple architectures that were previously used for combinatorial optimization, Structure2Vec [10] and ChebNet [12], as our starting point.

After making these initial decisions, we formulated our work plan. We present its objectives in the following section and its results in following chapters.

2.2 Objectives

We set the implementation of the infrastructure for model training and evaluation as our first objective.

With this infrastructure in place, we can then explore variants of GNNs and their performance in regression and ranking tasks on various graph families. With this approach we hope to obtain the knowledge necessary for training the best regression or ranking model and these experiments should also answer the question whether GNNs can learn to detect cliques.

Additionally, we would like to analyze the capabilities of our networks, investigate whether they can generalize on graphs of different structure and size, and possibly interpret their learned knowledge.

We do not expect to create a better practical algorithm for finding cliques in this work, but our work should serve as a proof of concept that GNNs can be used to reduce the number of calls in branch and bound algorithm. To achieve this, we first have to implement B&B capable of using neural networks and then we have to evaluate its performance using different heuristic functions.

Chapter 3

Empirical Evaluation of Network Variants

We present the main body of our work in this chapter, in which we focus on training and evaluation of neural networks designed for regression and ranking tasks. In regression task, the goal of the task is to predict values of $\omega(v)$ for all vertices of a graph and in ranking task the goal is to assign real valued scores to all vertices, so that vertices sorted according to these scores form the same sequence as if they were sorted according to $\omega(v)$ values.

In Section 3.1 of this chapter we describe our experimental framework, which we used for training and evaluation of neural networks, and we devote subsequent sections to specific experiments. In Section 3.2 we train models to solve the regression task, in Section 3.3 we evaluate models on ranking task and finally, we describe our generalization experiments in Section 3.4.

3.1 Framework for Training and Evaluation of Models

Before moving towards concrete experiments, in this section we describe multiple components, which we use in all our experiments. We begin by introducing the architecture of our GNN-based models, and then we describe our baseline models and the process of model training.

Note that the implementation of our models and training and evaluation infrastructure, as well as the code necessary to reproduce our results can be found by following the link in Appendix B – Implementation.

3.1.1 The Architecture of Models

We can formally view a graph neural network as a function $g_{\theta_1} : G \mapsto (h_{v_1}, \dots, h_{v_{|V|}})$ parameterized with θ_1, as we mentioned in Section 1.3.2.

Another neural network, in our case a multilayer perceptron, is then used to map these embeddings to final predictions, either in form of clique sizes or ranking scores:

$$m_{\theta_2} : \mathbb{R}^k \to \mathbb{R}$$

If we refer to all trainable parameters of these networks as $\theta = (\theta_1, \theta_2)$, the function expressed by our neural network (a composition of GNN with MLP applied to each embedding) can be defined as $F_\theta(G) = \Big((F_\theta(G))_1, \dots, (F_\theta(G))_{|V|} \Big)$, where

$$(F_\theta(G))_i = f_\theta(v_i) = m_{\theta_2}((g_{\theta_1}(G))_i)$$

In our implementation, the first graph convolutional layer $l_1 : \mathbb{R} \to \mathbb{R}^k$ maps the node tag (which is always set to 1 for all vertices) to the initial embedding and higher layers $l_i : \mathbb{R}^k \to \mathbb{R}^k, i > 1$ process the embeddings without changing their dimensionality.

We use two architectures of GNNs that are based on two types of graph convolutional layers we described in Section 1.3.3: an architecture of Structure2Vec to which we will refer as *S2V*, and an architecture we call *ChebNet*, which consists of a sequence of Chebyshev convolutional layers with parameter $K = 2$ (aggregation through the 1-neighbourhood of each vertex in each layer).

Similarly to convolutional layers, all layers of MLP, except for the output layer, represent functions of type $\mathbb{R}^k \to \mathbb{R}^k$ with fixed value of $k = 64$ and we interleave all convolutional and fully connected layers with leaky rectified linear activation function, $\text{leaky\_relu}(x) = 0.01 \cdot \min(x, 0) + \max(x, 0)$.

We describe the specifics of output layers and loss functions in Sections 3.2 and 3.3.

3.1.2 Baseline Models

To determine whether our networks are able to learn some structural patterns helpful for clique prediction, we also evaluate the validation and testing metrics using baseline models. We use specific baselines for each task.

Mean baseline model is parametrized with only one scalar and learns the mean $\omega(v)$ for all vertices of all training graphs.

Mean-per-degree baseline (or simply degree baseline) model learns a separate mean $\omega(v)$ for all vertices with a specific degree and thus uses at most as many parameters as the maximum degree of the training set. For vertices with degrees that were not observed in training this model predicts 0.

Rank-by-degree baseline simply outputs the degree $|N(v)|$ of each vertex v. This model has no trainable parameters and is only usable in the ranking task where its behaviour corresponds to sorting all vertices according to their degree.

Rank-by-degree-and-density baseline has also no trainable parameters and outputs a pair $\left(|N(v)|, \sum_{u \in N(v)} |N(u)|\right)$, so the vertices are ranked primarily on their degree and in case of ties, the density of their neighbourhood decides.

Note that degree baselines were specifically designed to find out, whether our networks use more complex information than just the degree of each vertex, resp. the sum of degrees of their neighbours.

3.1.3 Training Data Generation

There are two standard benchmark datasets that are commonly used to evaluate algorithms for MCP [49]. The dataset from The Second DIMACS Implementation Challenge: 1992-1993 [21] contains 80 graphs of 13 types with $100 - 4\,000$ vertices. These graphs include random graphs with a fixed probability for each edge, real-world problem graphs and graphs designed to make the search for maximum clique hard. The BHOSLIB dataset [51] was created by transforming hard SAT instances that model binary constraint satisfaction problems and contains 36 graphs with $450 - 4\,000$ vertices.

For training and validation purposes we generated random graphs of multiple types selected from DIMACS and BHOSLIB datasets. A graph type was included to the dataset if the generating procedure was simple to implement or if we were able to find an existing implementation of the generator.

We use the following graph families in our datasets:

- random graphs `C<N>.5`, `C<N>.9` with N vertices and edge probabilities of $0.5, 0.9$ respectively,
- `brock<N>` graphs that were designed to hide the maximum clique among the vertices with relatively low degrees; $\omega(G)$ is sampled uniformly from range $\{3, 4, \ldots, N\}$ as a parameter for `brock` graph generator,
- `dsjc<N>` k-partite random graphs with k sampled uniformly from range $\{2, 3, \ldots, N\}$ that mimic `C<N>.5` graphs but contain at least one k-clique,
- and small BHOSLIB graphs `rb<V>-<D>` that represent random hard constraint satisfaction problems with V variables each with a domain of size D.

We were also considering to generate Hamming graphs `hamm<B>` from DIMACS dataset, where vertices correspond to 1–B-bit code-words and two vertices are connected if they have Hamming distance at least d. But since Hamming graphs are regular, we did not include them into our training sets as GNNs are known to be unable to process regular graphs (Section 1.3.4).

When the generation process is completed, we label each vertex v of each generated graph G with the true value of $\omega(v)$. We use our own implementation of B&B algorithm that uses coloring heuristic functions and produces a sequence $\omega(v_1), \ldots, \omega(v_{|V|})$ for each graph.

We will provide the details of generated datasets, such as the number of generated graphs, with individual experimental results.

3.1.4 Training Setup

In our experiments we either use grid search to tune hyper-parameters (number of GNN and MLP layers, batch size and the initial learning rate) or we use architectures that proved successful in previous experiments.

Before training, we split each dataset into training and validation set by splitting the set of graphs of each type in 80 : 20 ratio.

Each model is then trained on the training set using early stopping with at most 200 epochs and the patience of 10 epochs. The learning rate is reduced by the factor of 0.2 if no improvement of validation loss is observed for 5 epochs.

All our networks are optimized using Adam optimizer [26].

3.2 Solving the Regression Task

In regression task we want to train the network to produce predictions $f_\theta(v)$ such that: $\forall v \in V : f_\theta(v) \approx \omega(v)$ and we can achieve this by minimizing the mean square loss on the training set $\mathcal{G}_T$:

$$l_{MSE}(\theta, v) = \left(f_\theta\left(v\right) - \omega\left(v\right)\right)^2$$

$$L_{MSE}(\theta, \mathcal{G}_T) = \frac{1}{|\mathcal{G}_T|} \sum_{G \in \mathcal{G}_T} \frac{1}{|V_G|} \sum_{v \in V_G} l_{MSE}(\theta, v)$$

The square loss function focuses on minimizing the worst errors and penalizes smaller errors less than absolute L1 loss $|f_\theta\left(v\right) - \omega\left(v\right)|$.

3.2.1 Output Layer and Target Transformations

We consider the simplest and most straightforward variant of the network with linear output layer and $\omega(v)$ targets as our first variant, which we will refer to as *absolute*.

Since it might be hard to train neural networks to produce estimates of unbounded clique sizes $\omega(v) \in \mathbb{N}$, we considered one more variant which maps the targets to $[0, 1]$ interval before training. In *relative-to-degree* variant we transform the output of MLP with logistic sigmoid $\sigma(x) = 1/\left(1 + e^{-x}\right)$ and we modify training targets to values relative to vertex degrees:

$$\omega_{deg}(v) = \frac{\omega(v)}{|N(v)| + 1}$$

To compare the effects of ω and ω_{deg} targets and also to achieve easier interpretability of loss function values, even if we train the network using relative targets, we convert its predictions to absolute targets $\omega(v)$ to compute the validation and testing mean square error (MSE).

While these targets achieve that the expected outputs of the network are bounded within $[0, 1]$ interval, and so the network should be easier to train and maybe even generalize better, they can also affect additional aspects of training and inference.

One such effect is that ω_{deg} penalizes prediction errors on vertices of lower degree more than errors on vertices of higher degrees. In detail, absolute prediction error of 1 for a vertex of degree 4 (resulting in loss $(1/(4+1))^2 = 1/25$), will be penalized with four times greater loss than than the same error made for a vertex of degree 9 (with loss of $(1/(9+1))^2 = 1/100$). This effect might decrease the ability of the network to distinguish cliques of similar sizes in vertices of relatively high degree.

In addition to ω and ω_{deg}, we considered one more output variant, in which the targets are transformed to values relative to $\omega(G)$, but due to theoretical reasons we decided not to study it in greater detail. We provide the definition, analysis and limited evaluation results for this variant in Appendix C – Target Transform ω_{mc}.

3.2.2 Model Training, Tuning and Initial Evaluation

To evaluate the performance of absolute and relative-to-degree variants of S2V and ChebNet networks, we generated 80 training, 20 validation and 100 testing 20-vertex graphs of each graph type mentioned in Section 3.1.3.

Using a grid search we trained multiple architectures of each of four variants with 3-5 graph convolutional layers and 2-4 MLP layers. The dimensions of the vertex embeddings and of the hidden layers were set to 64 and starting learning rates of $10^{-2}, 10^{-3}, 10^{-4}$ and batch sizes of 8, 16, and 32 graphs were used.

We present the testing mean square error achieved by our models and baselines in Figure 3.1 and in the first row of Table 3.1. Our models outperform baseline models, while S2V performs better with absolute targets and ChebNet achieves the globally best results with bounded targets.

Best Values of Hyper-Parameters and Stability of Training A subset of multiple models with similar hyper-parameters achieved similar results for each variant of GNN and target transformation. Therefore, we can use the results from the tuning process not only to find the best combinations of hyper-parameters, but also to explore the stability of the training process without the need to repeat the training multiple times.

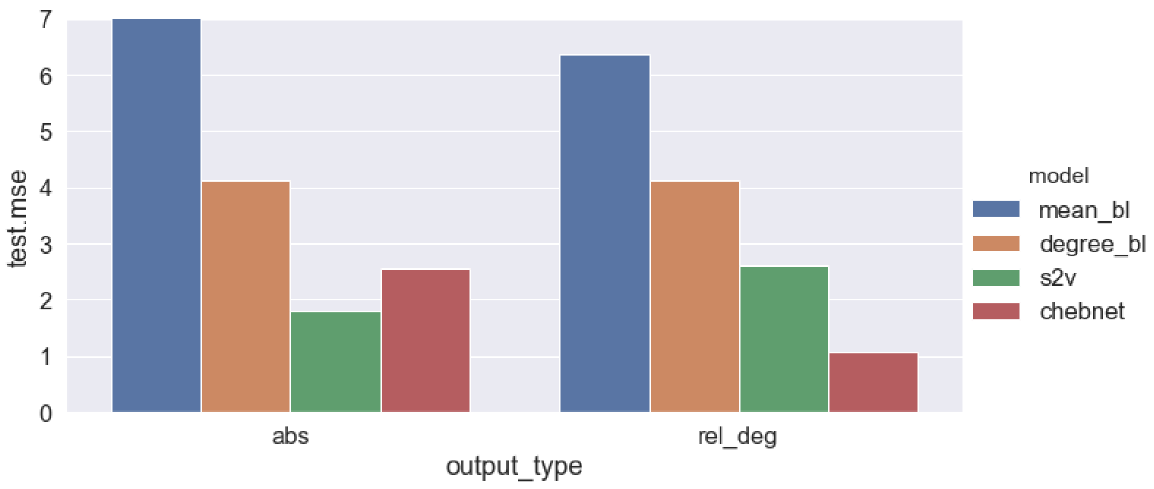

Figure 3.1: MSE for predictions of $\omega(v)$ achieved with two baseline models (mean, mean-per-degree) and two GNN-based models (S2V, ChebNet) trained with two variants of targets (absolute, relative-to-degree). Note that the mean baseline model achieves MSE of 16 when trained on absolute targets.

Variants of S2V model with 5 iterations of message passing showed much higher variance in MSE than variants with 3 or 4 iterations. This unstable behaviour might be a result of iterated multiplication of the weight matrix, which is shared across S2V layers. When S2V models were trained with batch sizes smaller than 32 and learning rate lower than 0.01, they achieve MSE of 2-3.5 on absolute targets and of 3-4 on relative targets, and thus consistently outperform baseline models.

The fact that all but two of the trained variants of ChebNet on absolute targets performed worse than mean-per-degree baseline suggests that ChebNet does not seem to train well on these targets.

In contrast with S2V model, as each of ChebNet layers uses a separate set of parameters, increasing number of convolutional layers improves the performance of ChebNet and variants of ChebNet with 5 convolutional layers trained on relative targets with learning rate 0.001 achieve mean square errors of 1-2. Furthermore, these errors can be systematically decreased using smaller batch sizes.

Overall, the training results seem to be more stable for ChebNet models.

3.2.3 Performance of Model Variants on Graph Families

To gain insight into what our models learned, we selected the trained instance that achieved the lowest validation mean square error for each combination of GNN and target transformation and we evaluated these models separately on graphs from each graph family. We trained new baseline models using exclusively graphs from particular graph families (but different from testing graphs) for this evaluation. We present the results of this evaluation in Table 3.1.

Table 3.1: MSE evaluated on all graphs and separately on 5 graph types (rows) using 4 models with 2 target transform variants (columns). Note that while GNN models (S2V, ChebNet) were trained on the whole dataset, baseline models (mean, mean-per-degree) were trained on specific graph types, except for baselines whose results are reported in the first row. The average MSE of 5 mean-per-degree models trained on separate graphs is 2.54.

	absolute				relative-to-degree			
	mean	deg	s2v	cheb	mean	deg	s2v	cheb
all	16.34	4.12	**1.80**	2.55	6.36	4.12	2.60	**1.06**
C.20.5	0.58	**0.37**	0.46	1.32	0.67	**0.37**	0.52	0.60
C.20.9	1.67	1.35	**1.20**	1.84	1.43	**1.35**	1.86	1.43
brock20	26.70	6.55	5.29	**5.07**	7.63	6.55	7.34	**1.80**
dsjc20	10.51	4.26	**1.42**	3.67	5.07	4.26	2.16	**1.09**
rb5-4	0.22	**0.17**	0.64	0.84	0.55	**0.17**	1.13	0.40

Description of Results S2V+absolute model performs comparably to mean-per-degree baseline on most graph types, with the most significant improvement achieved on `dsjc` graphs and a smaller improvement on `brock` graphs. This model fails to predict the clique sizes for `rb` graphs (mostly of sizes 4 and 5), which results in worse performance than mean baseline on these graphs.

ChebNet+absolute model achieves worse results than mean baseline on most graphs with exception of `brock` and `dsjc` graphs where it can outperform the degree baseline.

S2V+relative-to-degree model performs comparably to baselines on most graph types while it produces better predictions for `dsjc` graphs and worse predictions for `rb` graphs than baseline models. Notice, that this model achieves worse performance than S2V+absolute model on all graph families, so the target modification does not seem to help S2V model. Also note that while the mean baseline model is still parametrized with a single parameter, its predictions scale linearly with the degree, since it is trained on relative labels and its predictions are reverse-transformed into $\omega(v)$ format as $f_\theta(v) \cdot (|N(v)| + 1)$. Thus the mean+relative-to-degree baseline model achieves results comparable to mean-per-degree baseline.

ChebNet+relative-to-degree significantly outperforms baseline and previous models on `brock` and `dsjc` graphs and achieves results comparable to baseline models on other graph families.

Effects of Graph Family Specifics on Model Performance To interpret the results achieved on specific graph families, we provide selected statistical information about graph families in figures 3.2 and 3.3.

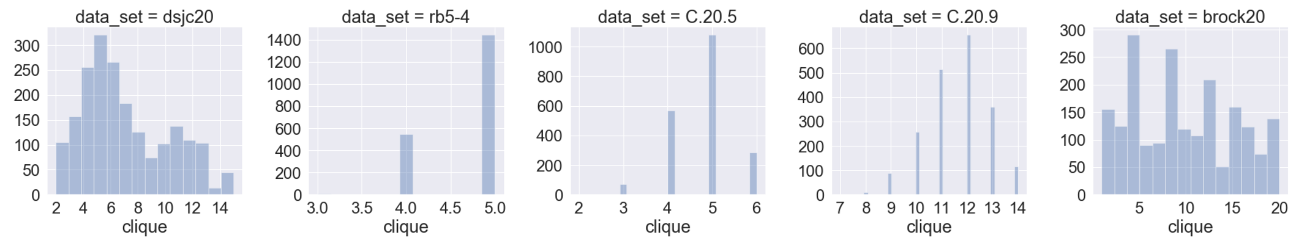

Figure 3.2: Distribution of $\omega(v)$ values for all vertices of 100 random 20-vertex graphs (2000 vertices per plot) for each graph family.

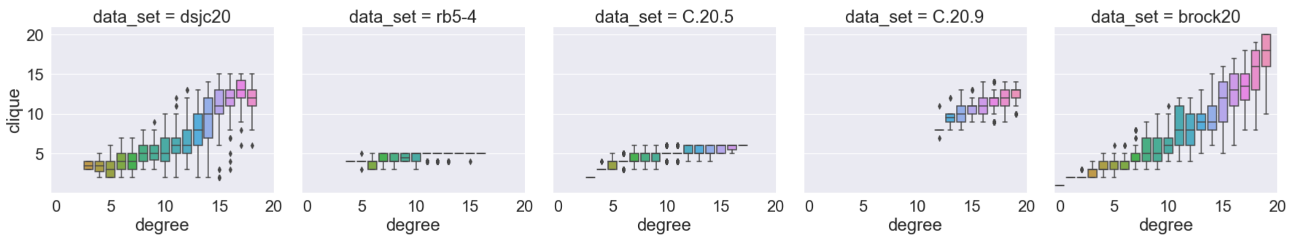

Figure 3.3: Distribution of $\omega(v)$ values for all vertices of 100 random 20-vertex graphs conditioned on vertex degree each graph family. In each boxplot, box at degree d represents a distribution of $\{\omega(v) \mid |N(v)| = d\}$.

As the distribution of clique sizes in random graphs `C.20.5` and `C.20.9` has relatively low variance and minimal variance in `rb5-4` graphs (see Figure 3.2), the clique sizes are well predicted by mean baseline. Furthermore, as clique distributions conditioned on degree (see Figure 3.3) have also low variances, these errors are reduced by the mean-per-degree baseline.

On the other hand, clique sizes in `brock20` and `dsjc20` graphs are distributed more uniformly, thus the mean baseline does not perform well on these graphs. Clique sizes are correlated with vertex degrees in theses graphs, so the errors are significantly reduced by mean-per-degree baseline, but since the clique size distributions conditioned on degrees have relatively large variances, the error of mean-per-degree baseline on these graphs is still much higher than on `C.20` and `rb` graphs.

Even though our models were trained on a mixed dataset containing all graph types, our best models show performance comparable to baselines trained on specific graph families on C.20 and `rb` graphs. This might tell us that our models either learn to distinguish graph families and learn the mean value (or possibly a set of parameters) for each of these families, or they are able to exploit more general structural features than just vertex degrees.

Furthermore, the difference in performance between baseline models and GNN models on `brock` and `dsjc` graphs shows that our models are able to base their predictions on more complex structural patterns than vertex degrees in these graphs.

3.2.4 Comparison of Performances of S2V and ChebNet

From the previous observations we can conclude, that ω_{deg} target transformation improves the performance of ChebNet models and hinders the performance of S2V models. Furthermore, ChebNet models generally do not train well on absolute targets but can achieve the globally best results for `brock20` and `dsjc20` graphs and can learn the behaviour similar to baselines on other graph families.

We can explain these phenomena by analyzing convolutional layers of our networks.

S2V layers directly sum the embeddings in their neighbourhood, so their predictions can scale well with increasing degrees, which explains relatively good ability to predict absolute clique sizes that are correlated with degrees. On the other hand, worse performance on relative targets might show that the accumulation effect is relatively hard to suppress in S2V architecture.

Vertices in ChebNet layers normalize the embeddings of their neighbour vertices before summing them together, which might result in its inability to accumulate large enough output values and to predict absolute clique sizes. Prediction of relative clique sizes in $[0, 1]$ interval seems to be a simpler task for ChebNet, as it does not require the model to produce large values.

The superiority of ChebNet over S2V can be a product of multiple differences in architectures. Since S2V layers share a single set of parameters, this model has potentially lower capacity than ChebNet and the repeated matrix multiplication might lead to a chaotic behaviour and to the problem of exploding gradients known from recurrent networks [17, Chapter 8]. Furthermore, S2V does not use previous local embedding $h_v^{(i-1)}$ in computation of $h_v^{(i)}$ and thus the information accumulated in S2V might be limited to the node degree after the first iteration, the sum of degrees of neighbouring vertices in second iteration and so on. S2V architecture might be thus very limiting and Dai. et al. [11] could have probably achieved much better results with use of more advanced GNN.

3.2.5 Conclusions and Limitations of the Experiment

With the first set of experiments we gained more insight into graph families we use, information about performance of baseline and GNN-based models and about differences between S2V and ChebNet models.

While clique sizes in `C.20.5`, `C.20.9` and `rb5-4` graphs are well predicted by baseline models and more sophisticated models do not outperform these simple models, clique sizes in `brock20` and `dsjc20` graphs are harder to predict even if the baseline models are aware of vertex degrees. Our models were able to use more complex structural patterns to predict clique sizes much more precisely than baseline models in these graphs.

We achieved the best results with ChebNet model trained on transformed targets and we explained its superiority over S2V model.

While we observed a relatively good performance of our models on `brock20` and `dsjc20` graphs, we still do not know which graph patterns are exploited by GNNs and a more detailed analysis of these results might be interesting.

Although we have seen that our models trained on mixed dataset do not outperform baselines on random and `rb` graphs, it might be also interesting to find out whether they can outperform these baselines if separate models are trained on graphs of each type.

Keep in mind that finding cliques in 20-vertex graphs might be easy not only because of the sizes of the instances, but these graphs might only contain a limited variety of substructures and correlations between patterns, e.g. between degrees and clique sizes might be much stronger and carry more information than in larger graphs.

For graphs with cliques of similar sizes, e.g. for `rb` graphs, it might be easy to roughly guess the clique size based on vertex degree, but this does not necessarily mean that B&B algorithm can easily find the maximum clique in these graphs. In contexts of exact B&B and non-exact heuristics, the ranking task may be more important and more practically useful than the regression task, since a good branching model can lead to fast discovery of large cliques.

We address these issues with experiments described in sections 3.3 and 3.4.

3.3 Learning to Rank

In ranking task, our goal is to predict scores $f_\theta(v)$ for vertices of $G = (V, E)$

$$v_1, v_2, \ldots, v_{|V|} \in V, \omega(v_1) \geq \omega(v_2) \geq \cdots \geq \omega(v_{|V|})$$

so that

$$f_\theta(v_1) \geq f_\theta(v_2) \geq \cdots \geq f_\theta(v_{|V|})$$

In order to evaluate the performance of our models on the ranking task, we first define an easily interpretable ranking metric and then we evaluate this metric for regression models from the previous section and we also train new variants of models using ranking margin loss.

3.3.1 Ranking Metrics

The most important ability of branching rule in B&B is distinguishing larger cliques from smaller cliques. Moreover, as long as all vertices of small cliques are assigned sufficiently low ranking scores, even if ranking scores in smaller cliques are imprecise, it has negligible impact on the task of finding large cliques.

Following this reasoning and also for the sake of simplicity, we distinguish only two types of vertices. We partition vertices into those that are contained in maximum cliques and those that are only contained in smaller cliques. Formally, we will call the vertices of $v \in V$ where $\omega(v) = \omega(G)$ *relevant*, and the vertices where $\omega(v) < \omega(G)$ *irrelevant*.

If we denote as $r(G)$ the number of all relevant vertices of a graph, $r(G) = |\{v \in V \mid \omega(v) = \omega(G)\}|$, we defined the *fraction-of-max-clique* (FMC) metric as the fraction of relevant vertices in first $r(G)$ highest ranked vertices. This metric is easily interpretable and accounts for the fact that different graphs contain different amounts of relevant vertices – to improve this metric by 10%, a model has to assign sufficiently higher scores to 10% vertices in maximum cliques.

We also considered common ranking metrics *precision@k* and *average precision* [48].

Precision@k, which stands for the fraction of relevant elements in the first k highest ranked predictions, is easily interpretable and focuses on the highest ranked vertices – those that will be selected first in B&B and in heuristics.

Average precision, which focuses on general ordering ability of models and is thus harder to interpret, is defined as $\sum_{k=1}^{|V|}(\text{precision@k} \cdot \text{rel}(k))/r(G)$, where $\text{rel}(k)$ is 1 if the vertex with predicted rank k is relevant and 0 otherwise. This metric corresponds to area under precision-recall curve used in classification tasks.

3.3.2 Ranking Loss Function

Specifically for the ranking task we trained new, *ranking* variants of GNN-based models with linear output layers and with use of a variant of margin ranking loss. The ranking loss for a pair of vertices in a graph, resp. for the whole dataset $\mathcal{G}$ can be defined as

$$l_{RANK}\left(\theta, v_i, v_j\right) = \left[\omega\left(v_i\right) > \omega\left(v_j\right)\right] \cdot \max\left(0, f_\theta(v_j) - f_\theta(v_i) + \text{margin}\left(v_i, v_j\right)\right)$$

$$L_{RANK}\left(\theta, \mathcal{G}\right) = \frac{1}{|\mathcal{G}|} \sum_{G \in \mathcal{G}} \sum_{v_i, v_j \in V_G \times V_G} l_{RANK}\left(\theta, v_i, v_j\right)$$

With zero margin, this loss function is only in effect for vertices where both $\omega\left(v_i\right) > \omega\left(v_j\right)$ and $f_\theta\left(v_i\right) < f_\theta\left(v_j\right)$ hold for a pair of vertices v_i, v_j. Since this loss can be easily minimized by outputting a constant value $f_\theta(v)$, a small non-negative margin is added to push the desired ranking scores for vertices with $\omega(v_i) \neq \omega(v_j)$ apart. To give more training information to our networks, we used margins based on difference of $\omega(v_i) - \omega(v_j)$:

$$\text{margin}(v_i, v_j) = 0.1 \cdot (\omega(v_i) - \omega(v_j))$$

Training and Tuning Ranking Models We trained and tuned hyper-parameters of rank variants of S2V and ChebNet models with the identical procedure and the dataset we used to train regression models in Section 3.2.2.

Similarly as in the regression task, lower number of convolutional layers improves the performance of S2V models and the models with 3 convolutional layers and initial learning rate 0.001 achieve relatively stable values of validation FMC 0.91-0.92. ChebNet models with 5 convolutional and 3 hidden MLP layers and initial learning rate 0.01 achieve validation FMC 0.92-0.93.

3.3.3 Evaluation of Ranking Abilities of Models

In this section we evaluate the best models trained with L_{MSE} (S2V+absolute, S2V+relative-to-degree, ChebNet+absolute and ChebNet+relative-to-degree with the lowest validation MSE) and the best models trained with the L_{RANK} (S2V+rank and ChebNet+rank with the highest validation FMC) on the ranking task using FMC metric. We evaluate this metric on 100 20-vertex testing graphs of each graph type.

We present the overall comparison of these models in Figure 3.4. Most prominent improvements over the baseline models are achieved by ranking variants of our models. While the less successful regression model S2V+absolute outperforms the rank-by-degree baseline, ChebNet+relative-to-degree model slightly outperforms rank-by-degree-and-density baseline. This suggests that even models trained for regression task might be relatively good at ranking vertices.

Performance on Graph Families Similarly as in the regression experiment, we also provide the FMC values evaluated separately for each graph type for each model. These results can be found in Table 3.2.

From regression models trained on absolute targets, the more successful model, S2V+absolute, achieves better ranking scores than ChebNet+absolute and achieves results similar to rank-by-degree-and-density baseline on all graph types. ChebNet+absolute achieves results closer to rank-by-degree baseline.

S2V+relative-to-degree model performs similarly to rank-by-degree baseline, but ChebNet+relative-to-degree model consistently outperforms baseline models on all graph types by 1-4% of FMC except for `C.20.9` graphs. The largest improvement over baseline models is achieved on `C.20.5` graphs. This model did not only achieve

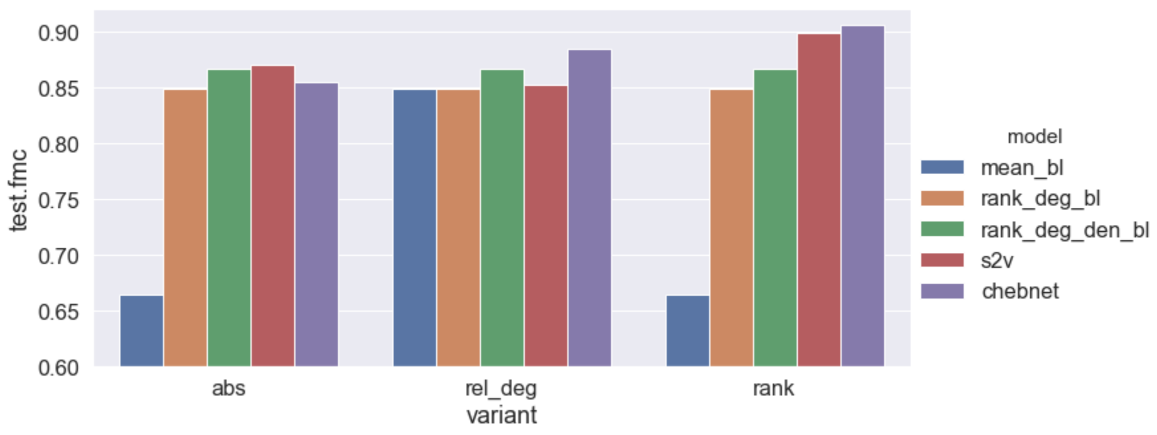

Figure 3.4: FMC achieved by 3 baseline models and 2 GNN-based models in three variants (absolute, relative-to-degree and rank). Notice that since ranking baselines only order vertices according to their degrees, resp. according to their degrees and local edge densities, they achieve the same FMC in all variants.

Table 3.2: FMC achieved by baseline models and variants of GNN-based models on all graphs (first row) and separately on graphs from each graph family. We highlighted the results of models that outperformed baselines and achieved FMC higher than the other GNN-based model of the same variant.

	baselines			abs		rel_deg		rank	
	mean	deg	deg_den	s2v	cheb	s2v	cheb	s2v	cheb
all	.664	.850	.867	**.871**	.855	.853	**.885**	.899	**.906**
C.20.5	.497	.732	.754	.748	.734	.728	**.792**	.821	**.827**
C.20.9	.793	.913	.931	.934	**.941**	.913	.930	.947	**.953**
brock20	.703	.895	.912	**.914**	.900	.891	**.921**	.923	**.932**
dsjc20	.603	.876	.901	**.912**	.877	.894	**.930**	**.949**	.946
rb5-4	.724	.833	.838	**.844**	.824	.837	**.851**	.858	**.873**

the best regression results, but also achieves best ranking results among regression models.

Finally, models trained with rank margin loss consistently outperform baselines and regression models on all graph types with notable improvements of 7% and 4.5% of FMC on `C.20.5`, resp. `dsjc20` graphs. Furthermore, ChebNet+rank model achieves slightly better results than S2V+rank on almost all graph types.

Additional Ranking Metrics We observed both, precision@5 and average precision metrics, to show similar relative differences between evaluated models as FMC. The only significant disparity was caused by precision@5 metric evaluated on random graphs with 0.9 edge density, which reported equal scores close to 1 for all models (except for mean baseline that orders vertices randomly, which achieved score of 0.8). This might be explained by observing that in `C.20.9` graphs, a few vertices with exceptionally high degrees are contained in almost all cliques and thus also in largest cliques in these graphs. Therefore, a small number of vertices of maximum clique can be easily found by ordering vertices according to their degrees.

Explanation of Observed Results Since the mean baseline model produces a constant value for each vertex, its behaviour is equivalent to ordering vertices randomly in the context of the ranking task. Therefore the FMC scores of mean baseline models are well explained by the fraction of relevant vertices $R(G)/|V|$ in each graph family, as shown in in Table 3.3.

Table 3.3: FMC achieved by mean baseline model compared to fraction of relevant vertices in each graph family.

graph type	C.20.5	C.20.9	brock20	dsjc20	rb5-4
testing FMC	.497	.793	.703	.603	.724
average $R(G)/\lvert V_G\rvert$	.511	.799	.688	.586	.727

Furthermore, clique sizes are correlated with degrees in our graphs, as was shown with boxplots in Figure 3.3 in Section 3.2.3. Therefore, ranking baselines, which order vertices according to their degrees, achieve improvements of 10-30% FMC over random vertex orderings produced by mean baselines.

Our ranking models achieved best improvements over degree-based baseline models in `C.20.5` and `dsjc20` graphs.

As these graphs contained the lowest fractions of relevant vertices, we considered it as the explaining factor in our first hypothesis. To test it, we created graphs with lower $R(G)/|V_G| \approx 0.3$, but although mean baselines showed worse performance, FMC achieved by GNN-based models on these graphs was still close to FMC achieved by degree baselines.

We still do not know the exact reason, why our models perform better on `C.20.5` and `dsjc20` graphs, but since we are interested mostly in differences between degree baselines and GNN-models, the cause of this phenomenon should be related to the relationship between degrees and clique sizes. Most probably, most vertices of maximum cliques in these graphs do not have extreme degrees and so other patterns, which are closely related to MCP, have to be used to order these vertices.

3.3.4 Conclusions

With our ranking experiments we shown that models that successfully solve the regression task can be also used for ranking and that special ranking loss function can improve the ability of GNN-models to rank vertices of graphs according to their clique numbers $\omega(v)$.

The improvements over baseline models achieved by GNN-based models are relatively small, but consistent over all graph families. This suggests that our models might not outperform classical B&B branching heuristics on the tested graph families, but also that our models are capable of learning structural patterns and using degree information more efficiently than simple ranking baselines.

The main drawback of our experiment is the size of our graphs and the selection of testing graph types, as large clique vertices can be found by their large degrees relatively easily in all graph types. In the future it might be interesting to train or evaluate our models on graphs where the vertex degrees are not correlated with clique sizes.

A recurring theme of our experiments is that some instances are easy to solve with simple approaches and thus the overall improvement of metrics seems less significant. In the future work, a good practice might be to split the testing instances into those that are well solved by simple heuristics and into harder ones.

3.4 Generalization Experiments

In our last set of experiments, described in this section, we observe the abilities of our models to generalize on larger graphs and on graph types not seen during training.

3.4.1 Generalization on Size

To observe the generalization ability of our models, we generated 100 50-vertex graph of each graph type and we present the values of MSE and FMC achieved by baseline and GNN-based models in tables 3.4 and 3.5. Note that while we reuse GNN-based models trained on 20-vertex graphs in this evaluation, regression baseline models (mean and degree baseline model) were trained separately for each graph type using 50-vertex graphs different from testing graphs.

Regression Task The complete failure of S2V models to generalize on larger graphs in regression task is most probably a result of unnormalized summation of embeddings of neighbour vertices. ChebNet+absolute model manages to keep relatively small MSE on `C.50.5` and `rb9-6` graphs and MSE comparable to mean baseline on `brock50` and `dsjc50` graphs. Finally, ChebNet+relative-to-degree fails completely on graphs with

Table 3.4: MSE evaluated on all graphs and separately on 5 graph types (rows) using 2 baseline models and 2 GNN-based models with 2 target transform variants (columns).

	absolute				relative-to-degree			
	mean	deg	s2v	cheb	mean	deg	s2v	cheb
all	94.25	35.55	26641.12	76.18	54.92	35.55	445.98	99.05
C.50.5	0.42	0.34	1563.62	2.31	0.72	0.34	356.37	46.74
C.50.9	1.89	1.59	66231.35	64.55	1.77	1.59	585.17	197.08
brock50	192.90	50.62	42653.29	217.06	74.27	50.62	246.26	82.57
dsjc50	77.79	19.35	5652.08	98.99	39.29	19.35	234.86	23.08
rb9-6	0.28	0.24	17811.62	3.80	0.79	0.24	780.47	142.34

low clique size variance as its predictions are reverse transformed to $\omega(v)$ format as $f_\theta(v) \cdot (|N(v)| + 1)$, but maintains performance similar to dergee baseline on `brock50` and `dsjc50`, whose clique sizes are more correlated with degrees.

As regression tasks are usually very sensitive to the magnitude of inputs, these results are not unexpected. A better generalization ability could have been achieved, if we trained our models on graphs of various sizes, but since our models are probably not able to fully grasp the definition of cliques, this generalization would eventually fail at some point.

To conclude, the models we trained are not very useful for regression task on larger graphs as they produce large absolute errors on these graphs. The observed generalization abilities of ChebNet models are possibly more influenced by rough architectural features of its GNN and by target transformations than by the learned weights.

Table 3.5: FMC achieved by baseline models and variants of GNN-based models on all graphs (first row) and separately on graphs from each graph family. Again, we highlighted the results of models that outperformed baselines and achieved the FMC higher than the other GNN-based model of the same variant.

	baselines			abs		rel_deg		rank	
	mean	deg	deg_den	s2v	cheb	s2v	cheb	s2v	cheb
all	0.551	0.766	0.775	**0.782**	0.732	0.769	0.744	0.787	**0.803**
C.50.5	0.431	0.608	0.613	**0.627**	0.552	0.616	**0.623**	0.641	**0.665**
C.50.9	0.648	0.801	0.819	**0.820**	0.808	0.801	0.697	0.820	**0.830**
brock50	0.569	0.830	0.838	0.837	0.767	0.829	0.797	0.845	**0.857**
dsjc50	0.536	0.904	0.912	**0.930**	0.865	0.912	**0.929**	0.936	**0.963**
rb9-6	0.569	0.685	0.691	**0.694**	0.670	0.686	0.677	0.693	**0.702**

Ranking Task The ranking results show that our models maintain their advantages over baseline models when evaluated on 50-vertex graphs. Regression models achieve results comparable to baseline models with slight improvements in case of S2V+absolute models. ChebNet model trained with ranking loss outperforms baselines on all graph types with largest improvements of 5% of FMC on `C.50.5` and `dsjc50` graphs.

Since only the relative order of vertices is important in ranking task, it is less sensitive to magnitudes of $\omega(v)$ and $f_\theta(v)$. From this experiment we learned that even though our regression models fail to predict clique sizes in larger graphs, since they assign higher scores to vertices of higher degrees, they perform comparably to ranking baselines in the ranking task. These results also support our previous observation that ChebNet+rank model consistently outperforms baseline models and that our models exploit patterns in `C.50.5` and `dsjc50` more effectively than in other graph types for the ranking task.

It is quite interesting that even though S2V+absolute model completely fails in the regression task, it still achieves better performance than ranking baselines in the ranking task. This phenomenon might be possibly explained with an analogy to random walk processes: since cliques are the densest structures in graphs, a random walker that starts in a clique will have relatively high probability to stay within the clique after few steps. Therefore, ranking scores of S2V+absolute model, and possibly also scores of other models, can be results of exponentiation of adjacency matrix A, analogically to probabilities resulting from exponentiation of transition matrix in Markov processes. An interesting future baseline model could use values of $A \cdot \vec{1}$, $A^2 \cdot \vec{1}, \dots, A^n \cdot \vec{1}$ or similar values produced with laplacian matrix L as input features for a simple ML model.

3.4.2 Generalization on Structure

We performed the following experiment to observe the ability of our models to generalize on graph types unseen in training and also their ability to over-fit to a specific graph type.

First, we generated 500 20-vertex graphs of each type, a total of 2500 graphs and we split graphs of each type into 400 training graphs and 100 validation/testing graphs. If we consider graphs of each type as a separate dataset and all graphs in total as another dataset, we prepared 6 datasets `C.20.5`, `C.20.9`, `brock20`, `dsjc20`, `rb5-4` and `All` for this experiment.

In the second step, we trained a regression model ChebNet+relative-to-degree and a ranking model ChebNet+rank on each dataset, creating 6 trained instances of each model.

Finally, we evaluated MSE for each of 6 instances of the regression model on each of 6 datasets (using 100 testing graphs per graph type), and analogically we evaluated FMC for the instances of the ranking model.

In figures 3.5 and 3.6 we present the observed values of MSE, resp. FMC metrics in a form of heatmaps. We also present generalization errors that are caused by evaluating models using datasets different from training ones.

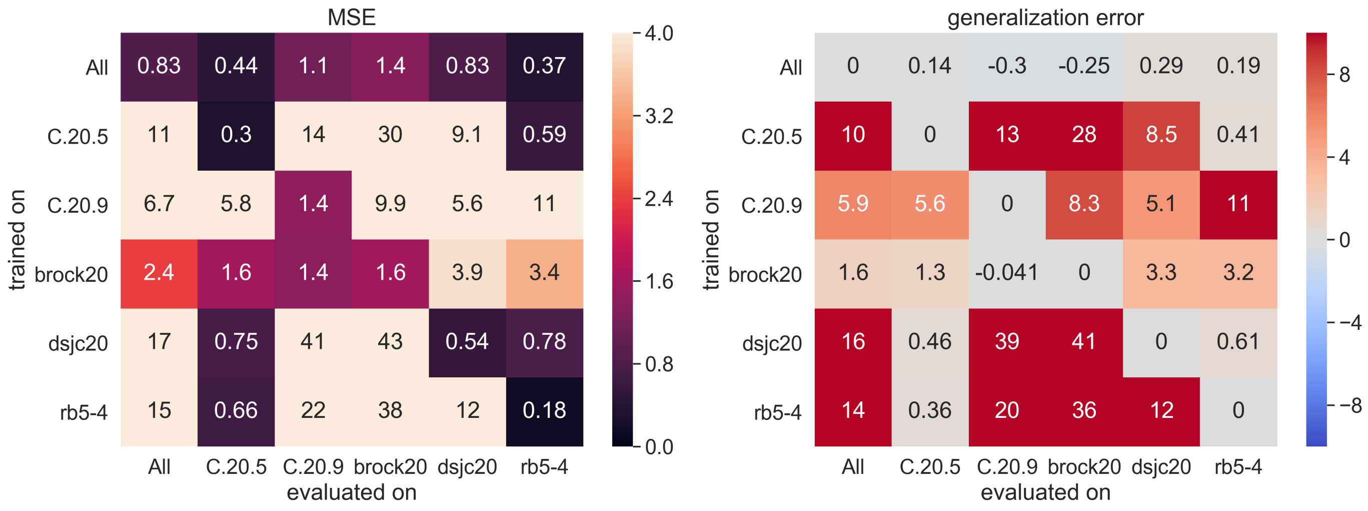

Figure 3.5: Left: MSE for 6 instances of ChebNet+relative-to-degree model (rows) achieved on 6 testing datasets (columns). Right: generalization error produced from the left heatmap by substracting diagonal values from each row, red values denote deterioration while blue values denote improvement.

Table 3.6: MSE achieved by degree baseline model on each dataset.

graph type	All	C.20.5	C.20.9	brock20	dsjc20	rb5-4
MSE of degree baseline	4.12	0.37	1.35	6.55	4.26	0.17

Regression Task The regression model trained on all graphs achieves similar performance as models trained on separate datasets. It seems that for `C.20.9` and `brock20` graphs, training on graphs of all types slightly improves the performance, while the best performance on `C.20.5`, `dsjc20` and `rb5-4` graphs is achieved by specialized models.

This phenomenon might be explained with an observation that a common property of `C.20.9` and `brock20` graphs is that they contain more cliques of larger sizes than other graph types, so the model trained on all graphs might overestimate the clique sizes in `C.20.5`, `dsjc20` and `rb5-4` graphs.

Since clique sizes are very limited in `rb5-4` graphs, model trained on `rb5-4` dataset can finally achieve the performance of the degree baseline (see Table 3.6).

Relatively good generalization ability of the model trained on `brock20` graphs might be explained by the fact that brock graphs contain graphs of various densities and clique sizes. This fact also explains, why other trained instances fail on brock dataset.

Our final observation is that `C.20.5` and `rb5-4` graphs are well predicted by `C.20.5`, `rb5-4` and `dsjc20` models, as all these graphs contain cliques of similar sizes (3-6). Also notice that `C.20.5` and `rb5-4` models fail to predict clique sizes in more complex `dsjc20` dataset, as it also contains larger cliques.

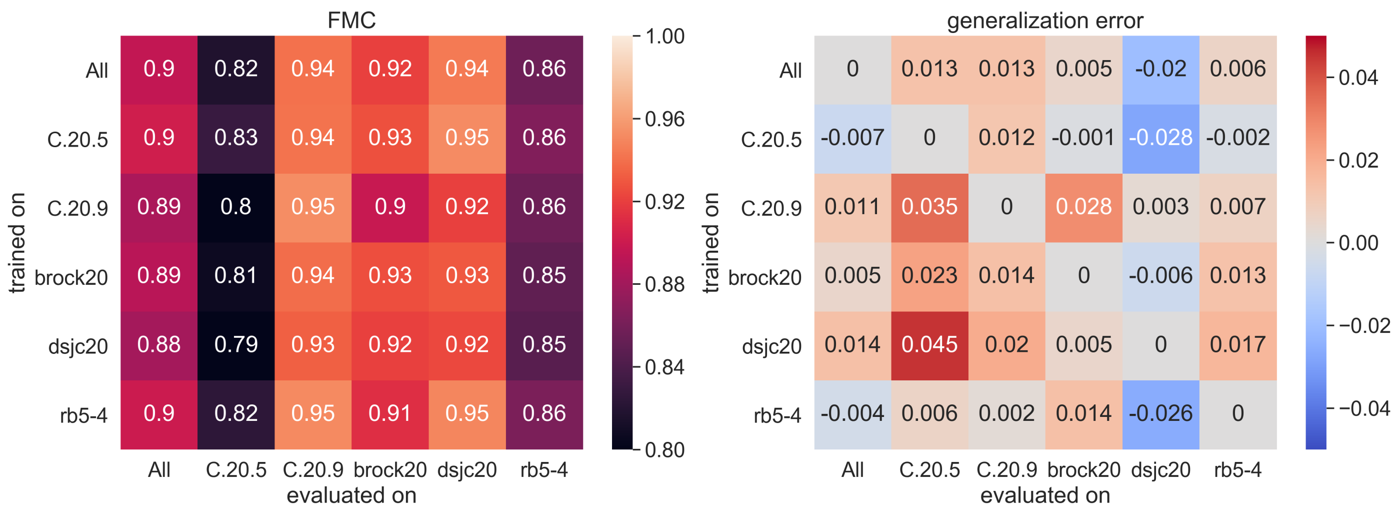

Figure 3.6: Left: FMC for 6 instances of ChebNet+rank model (rows) achieved on 6 testing datasets (columns). Right: generalization error produced from the left heatmap by substracting each row from the diagonal, red values denote deterioration while blue values denote improvement.

Table 3.7: FMC achieved by rank-by-degree-and-density baseline model on each dataset.

graph type	All	C.20.5	C.20.9	brock20	dsjc20	rb5-4
FMC of deg-den baseline	.867	.754	.931	.912	.901	.838

Ranking Task The fact that FMC values in the left heatmap in Figure 3.6 depend mostly on the evaluation dataset (specifically, on the fraction of vertices in largest cliques) might suggest that our ranking models use similar graph properties when they are trained on different graph types. Furthermore, as almost all instances of the ranking models outperform the rank-by-degree-and-density-baseline (see Table 3.7), their superiority over the baseline model is possibly an effect of a very simple mechanism, e.g. of ordering vertices according to a fitting definition of local density.

The model trained on `C.20.5` graphs seems to generalize well on all but `C.20.9` graphs and conversely, most of other models fail to generalize on `C.20.5` dataset. This is in line with our observation from Section 3.3 that vertices in `C.20.5` graphs are relatively hard to rank and thus model trained on this dataset learns the knowledge useful for the ranking task and other models fail on this dataset.

The relationship between `C.20.5`, `rb5-4` and `dsjc20` is also interesting in the case of ranking task, since the first two models seem to achieve the best performance on `dsjc20` graphs. `C.20.5` and `rb5-4` graphs only contain cliques of sizes 3-6 and therefore the improvements in ranking of `dsjc20` vertices happen most probably in graphs with relatively small maximum cliques.

3.4.3 Conclusion

The training of regression models in Section 3.2 brought us understanding of ChebNet and S2V GNN models and the ranking experiment in Section 3.3 showed us that training models using ranking loss can improve the ability of models to distinguish clique sizes.

Experiments on larger graphs mostly contained negative results and they lead us to a following question: "To what extent are the (ranking) abilities of GNNs simple consequences of their architectures and how significant is the effect of the training process?"

With experiments on structural generalization, we gained new insights into relationships between graph families. We showed that the graph families where our models achieved the best improvements over baseline models (`brock20` in regression task and `C.20.5` in ranking task), also seem to produce the models with the best generalization abilities and are hard to solve for models trained on other graph families. This observation outlines a possibly promising methodology for selection of families of training instances.

Chapter 4

Enhancing Branch and Bound

In this chapter we finally use our neural models as branching rules in B&B and we compare their performance to standard heuristic functions. We first describe simpler common branching and bounding rules in detail and then we present the results of the experimental comparison of these heuristic functions to our neural heuristics.

We described the computation of the branch and bound algorithm in the first chapter of our work. Feel free to read about its specifics in Section 1.1.3.

4.1 Heuristic Functions for B&B

The efficiency of B&B algorithm depends mainly on the quality of branching and bounding heuristic functions h_{br}, h_{bo}. The branching rule should order the vertices of the candidate set P from those contained in larger cliques to those contained in smaller cliques and the bounding rule should provide an upper bound on $\omega(v)$ in the subgraph induced by P. We will denote the size of the maximum clique in the subgraph induced by $\{v\} \cup N(v)$ in G_P as $\omega_P(v)$.

4.1.1 Degree-based Heuristics

The simplest way of bounding $\omega_P(v)$ is using the vertex degree, since in each graph $|N(v)| + 1 \geq \omega(v) \geq \omega_P(v)$. These bounds can be tightened by only considering neighbours of v within P. We will denote the degree of v in G_P as $\deg_P(v)$.

The vertex degree or $\deg_P$ can be also used as a branching heuristic and may work well in graphs where clique sizes are correlated with degrees. In addition to the selection of the next vertex v to add to the clique, the branching rule also defines, which vertices will remain in P in the next recursion call, since $P' := P \cap N(v)$. Therefore, visiting high-degree vertices first might also result in increased amount of overall work.

4.1.2 Coloring-based Heuristics

In the vertex coloring problem we search for the smallest $k \in \mathbb{Z}^+$ such that the vertices of graph G can be partitioned to k color classes where adjacent vertices belong to different color classes. We will denote the smallest such k, the *chromatic number* of G, as $\chi(G)$.

Since all clique vertices have to be assigned different colors, it is easy to see that $\chi(G) \geq \omega(G)$ and thus vertex coloring can be used in bounding heuristics. Currently, there are many approaches based on the idea of vertex coloring, but due to its simplicity, we selected the heuristic rule developed by Tomita et al. [42] as the representant of more advanced heuristics.

In their approach, a simple heuristic algorithm is used to assign color classes to vertices of P. These vertices are processed sequentially and each vertex is assigned the lowest color which maintains the feasibility of coloring: the lowest color that is not present among its neighbours. We use positive integers to represent color classes and we denote the class assigned to vertex v as $x_P(v) \in \mathbb{Z}^+$.

Notice that this heuristic does not ensure that $\forall v \in P : x_P(v) \geq \omega_P(v)$, but it guarantees that at least one vertex of the largest clique will be assigned sufficiently high color so that the largest clique is not pruned away. Formally, this approach guarantees that $\exists v \in P, \omega_P(v) = \omega(G_P) : x_P(v) \geq \omega_P(v)$. The ingenuity of this idea is that it ensures that B&B finds the optimal solution, but since it assigns lower values to most vertices in largest cliques, a large part of the search tree can be pruned away.

The authors used color classes both for bounding and for branching, $h_{bo}(v) = h_{br}(v) = x_P(v)$, but note that the coloring bounding rule can be combined with any branching rule. Contrary to what one may think, when a vertex v does not seem promising ($h_{bo}(v) + |C| \leq |C^*|$), it is not immediately removed from the candidate set P by B&B, but the bounding rule only decides that this vertex is not immediately added to the partial clique. As a result, all vertices of the largest clique can be retained in the new candidate set P'.

In our implementation, we use a slightly modified version of the coloring heuristic algorithm: we order the vertices in non-ascending order according to $\deg_P(v)$ before we assign their colors. This approach was mentioned by Konc et al. [28] to provide even tighter bounds. It was not directly used in their approach, as it requires the computation of G_P, however.

4.1.3 Optimal B&B Rules

As we mentioned in Section 1.1.3, with $h_{bo}(v) = h_{br}(v) = \omega_P(v)$, only $\omega(G)$ recursion calls are necessary to discover the maximum clique of G. Furthermore, all other branches of the search tree are pruned away using these rules and so no other calls need to be executed. This approach thus achieves the theoretical minimum for the number of B&B recursive calls.

To compare our results with this theoretical minimum, we implemented these rules by solving the MCP for each subgraph induced by $P \cap (\{v\} \cup N(v))$.

4.2 Neural Heuristic Function

The predictions of our neural networks can be directly used in B&B framework as the branching rule $h_{br}(v) = f_\theta(v)$ when absolute and ranking variants of models are used. When relative-to-degree variant is used, the predictions of the model are reverse transformed to correspond with absolute clique sizes, $h_{br}(v) = f_\theta(v) \cdot (\deg_P(v) + 1)$.

Implementation Notes The predictions for G_P are recomputed in each recursive call. In our current implementation, G_P is first stored in the filesystem, then the predictions $f_\theta(v)$ are computed by our model using the Python interpreter and finally, the predictions are passed to the B&B algorithm using another temporary file.

Since we focused on the proof of concept in our work, our implementation of B&B is mainly focused on code modularity and on gathering metrics, and we did not optimize it further. Python-C++ interface or C++ models might be used to improve the efficiency of the neural heuristic, however.

Complexity The computational complexity of neural inference on graph $G_P = (V_P, E_P)$ is $O(|V_{G_P}| + |E_{G_P}|)$, if we consider the the number of layers and the embedding dimension as constants.

In our implementation, however, the computation of neural heuristic function requires approximately 100-times more computation time than computation of simpler coloring heuristic when compared on 50-vertex graphs. Moreover, the effect of communication via filesystem becomes prevalent for smaller graphs and subgraphs G_P.

While the computation of the coloring, resp. of neural heuristic for 50-vertex graph requires 10^{-4}s, resp. 10^{-2}s, the overhead caused by the filesystem and the Python interpreter is 0.3×10^{-2}s.

Although it might be impractical to use these models for smaller instances, GPU parallelization might help these models to match or outperform classical algorithms in terms of computation time on large instances.

A more delicate approach might be also explored in the future, where we the costly neural predictions are not computed in all recursive calls of the B&B algorithm. The network could be either used only once, before the start of the B&B, or in particular recursive calls, e.g. when the candidate set is reduced to less than one tenth of its original size.

4.3 Evaluation of Heuristic Effectiveness

To compare the performance of heuristic functions, we generated 50 50-vertex graphs of each of six graph types (`C.50.5, C.50.9, dsjc50, brock50, hamm6, rb9-6`), a total of 300 graphs, and 100 100-vertex graphs of `C.100.5, dsjc100, brock100, rb13-8` types, a total of 400 graphs.

In our experiments, we use coloring and optimal rules described in sections 4.1.2 and 4.1.3 for bounding. We also tried using degree-based heuristics for bounding, but the number of recursion calls required by B&B with these heuristics was much larger than when coloring and optimal functions were used for bounding.

In combination with these bounding functions, we evaluate the performance of three baseline branching heuristic functions: degree branching heuristic based on $\deg_P(v)$, coloring heuristic based on $x_P(v)$, and optimal rule, which predicts $\omega_P(v)$ for each vertex.

Finally, we also evaluate two variants of neural branching functions, which use the predictions of ChebNet+relative-to-degree regression model and ChebNet+rank ranking model. We described the architecture and the training and tuning of these models in Chapter 3 and these models were trained on 20-vertex graphs of all types, with exception of regular hamming graphs.

Since the computation times of coloring, neural and optimal rules are currently incomparable, our first idea was to observe the number of recursion calls performed by variants of B&B. Using the assumption that the computational complexity of one recursion call is usually $O(|E_{G_P}|)$, we present the amount of *work* executed by variants of B&B instead, although we arrived to similar conclusions even when we observed the number of calls. If we denote as $\mathcal{P}_h$ the collection of candidate sets produced by recursion calls of B&B executed on graph G with heuristic h, the number of calls is simply $|\mathcal{P}_h|$ and we define $\text{work}(h, G)$ as $\sum_{P \in \mathcal{P}_h} |E_{G_P}|$.

4.3.1 Experimental Results

We present the results of the experimental evaluation for 50-vertex and 100-vertex graphs in figures 4.1 and 4.2.

Space for Branching Improvement In most graph types, the quality of the bounding rule seems much more important than the quality of the branching rule. In 50-vertex graphs, the ratio of work needed with coloring heuristic compared to work needed with optimal bounding rule achieves the order of 2, and in 100-vertex graphs this ratio is close to 10. Although the quality of the bounding rule is generally more important, there are some graphs where the improvement in branching rule can still improve the performance of B&B.

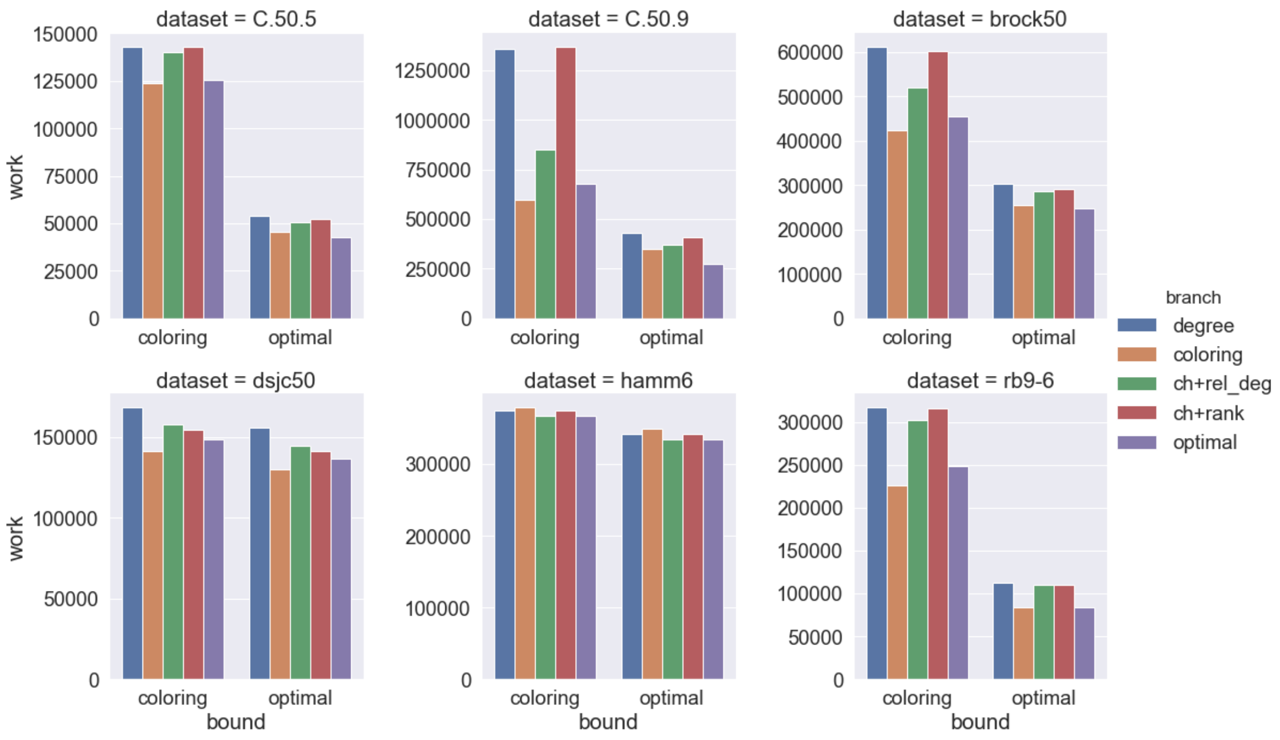

Figure 4.1: The total amount of work needed to find maximum cliques in 50 50-vertex graphs of 6 types (subplots). Each subplot contains two groups of results produced with coloring, resp. optimal bounding rule.

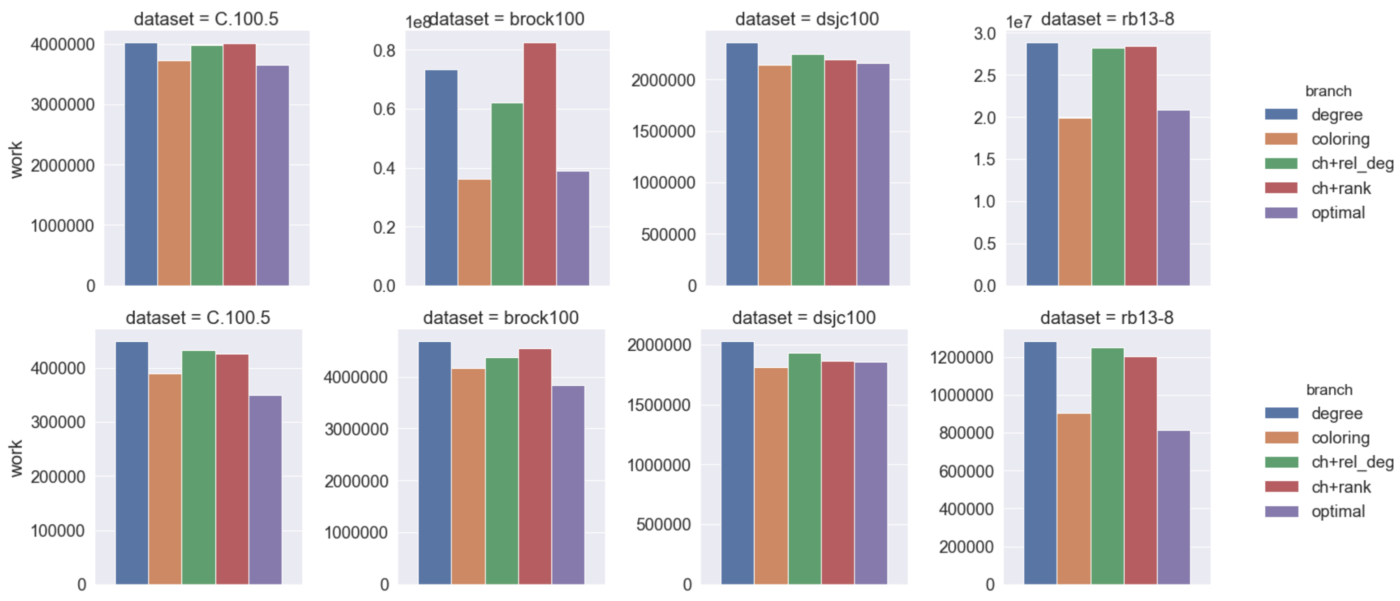

Figure 4.2: The total amount of work needed to find maximum cliques in 100 100-vertex graphs of 4 types (columns). Each column contains results produced with coloring (top), resp. optimal (bottom) bounding rule.

The most significant gap between the degree branching rule and the optimal branching rule is observed for `C.50.9` and `brock` graphs. In these graphs, regression ChebNet heuristic outperforms the degree heuristic, but fails to outperform the coloring heuristic. ChebNet+rank model achieves the performance comparable to degree heuristic in these graphs.

There is also a space for improvement of degree branching heuristic in `rb` graphs, but our models fail to outperform the degree baseline on these graphs.

In other graphs, `C.X.5`, `dsjcX`, `hammX`, the differences between degree and optimal branching rules are negligible.

Comparison of Regression and Ranking Model The fact that ChebNet+rank model does not achieve better performance than the regression model, might tell us that although our ranking metrics all reported similar results, they might not be relevant for the final task of branching in B&B. Possibly, a better approach than separating the training and evaluation process from the model's usage, would be to train models with a more direct contact with the algorithm. With B&B for example, models could be trained on graphs induced by candidate sets G_P gathered in the run of B&B and their performance could be evaluated using the final metrics, such as number of calls, the work performed or even the computation time of B&B.

Easily Colorable Graphs Since hamming graphs are regular graphs where all cliques are of the same size, no branching rule can improve the performance of B&B and its efficiency depends solely on the bounding rule. Furthermore, the coloring bounding rule achieves almost optimal performance on these graphs.

Another graph type, where the coloring heuristic provides almost optimal bounds are `dsjc` graphs. This is not unexpected, since these graphs were described by their authors as easily colorable with small number of color classes [22]. Notice that the coloring branching heuristic preforms less work than our implementation of optimal branching rule. This is caused the fact that vertices are ordered according to their degrees before colors are assigned, so the vertices with highest degrees are assigned the lowest colors and are thus less likely to be visited by B&B. This reveals that in order to achieve the minimum amount of work, the branching rule should order the vertices primarily based on $\omega_P(v)$ in descending order, but secondarily based on $\deg_P(v)$ in ascending order.

Limitations of the Coloring Bounding Rule We observed that even when neural or degree h_{br} led to faster discovery of larger clique than the coloring h_{br}, since it visited vertices with lower $x_P(v)$ first, the candidate vertices with higher colors that were considered later were not pruned away. On the other hand, the coloring branching heuristic always visits vertices with higher $x_P(v)$ first, so if larger clique is discovered, other candidate vertices with lower $x_P(v)$ are successfully pruned away.

This suggests that the coloring approach relies heavily on the coloring branching order and that neural or degree branching rules might possibly achieve better results if they are used in combination with bounding rule that is less tight, but does not demand a specific branching order.

Other possible solutions, how to avoid the problems that come with bounding rule, include using bounding rule which does not guarantee the optimality of the solution (e.g. also provided by a neural network) and using neural branching functions in heuristic algorithms other than B&B.

4.3.2 Conclusions

Across graph types, our models either provide branching rules comparable to or better than the degree baseline, but they still do not outperform simple but effective coloring heuristic.

The most notable improvement is achieved by regression ChebNet model on `C.50.9` and `brock` graphs. While we observed its ability to learn to predict clique sizes in `brock` in Section 3.2, the improvement on `C.50.9` graphs is unexpected. The achievements of our models in regression and ranking task on `dsjc` graphs are not applicable for the B&B algorithm, since MCP can be solved almost optimally using coloring heuristic.

The ranking model achieves worse performance than regression model on most graph types, which tells us that either the differences we observed when training ranking models are not significant enough to improve the B&B or the ranking metrics and graphs used in evaluation might not be relevant for the task of branching. Perhaps, better results can be achieved if the models are trained and evaluated in the context of the final task.

One last point to take home from these experiments is that to improve an algorithm using ML, one should select instances where there is at least theoretical space for improvement. Even if our results might seem negatively, deep learning approaches and GNNs might prove more effective in improving heuristic approaches.

Conclusion

In our work we investigated whether GNNs can be used to detect cliques and whether they can can be used as an effective branching heuristic of B&B algorithm.

Our results show that even relatively simple GNNs Structure2Vec and ChebNet can learn to exploit structural patterns more complex than vertex degrees and local densities to predict $\omega(v)$ and rank vertices according to $\omega(v)$. Concretely, ChebNet models, which use Chebyshev graph convolutional layers, significantly outperform mean-per-degree baseline on `brock20` and `dsjc20` graphs with various clique sizes and achieve performance comparable to mean baseline on graphs with low clique size variance. ChebNet+rank model achieves consistently better results than rank-by-degree-and-density baseline with the largest improvement on `C.20.5` graphs, where large-clique vertices cannot be easily found using only degree information. Furthermore, models trained on graphs where they showed the best improvement over baseline models seem to show the best generalization abilities in both tasks.

On the other hand, our size-generalization experiments suggest that these models are not complex enough to grasp the underlying concepts of cliques and that their architectural differences have a large effect on their performance and generalization abilities. We showed that our models fail to generalize on larger, 50-vertex, graphs in the regression task, but they can maintain their ranking abilities, which might be more a result of their architectural features (weigh matrix exponentiation, ω_{deg} targets) then the result of their learned weights. Since we found 20-vertex graphs very limiting in the context of MCP (with clique of size 10, a random ordering baseline would select a vertex of the maximum clique with probability 0.5), possibly, much better improvements might be achieved with more complex GNNs trained and evaluated on larger graphs selected from specific graph families where clique sizes are not correlated with vertex degrees or other simple local graph features.

While B&B using ChebNet+relative-to-degree model as its branching heuristic needs to perform less work than B&B guided with a simple degree heuristic, it does not outperform a more advanced coloring heuristic. Furthermore, since we found out that the quality of bounding function is probably much more important than the quality of branching function in B&B for MCP, the goal of improving the branching function can be pursued only if graphs with at least theoretical potential for improvement of branching rules are discovered.

While exact approaches to MCP have to review all possible cliques, heuristic approaches achieve their fast computation times by limiting themselves to only the most promising areas of the graph. In the future it might be interesting to utilize neural networks for fast discovery of sub-optimal solutions, for example by searching the tree not with a depth-first-search (employed by branch-and-bound), but with a more exploratory search strategy.

In addition to attending to our main objectives, we also produced multiple other assets and observations which we consider to be a valuable part of our contribution.

In the beginning of our work we reviewed literature on topics of MCP and B&B, machine learning for combinatorial optimization, methods for graph embeddings and GNNs. We consider the discovery of three independent formulations of GNNs we described in Section 1.3.3 as the most fascinating part of our review, since they connect GNNs to various research areas (random walks, graphical models, convolutional networks and spectral graph theory) which might inspire future research topics.

To make our work possible, we also had to implement the experimental infrastructure for graph generation and labelling, training and evaluation of models, visualization of results, and for using neural networks in B&B. We paid special attention to keep our code clean, modular and to make our research reproducible. The implementation of our infrastructure and experiments can be found by following the link in Appendix B – Implementation.

In our experiments, we considered multiple variants of models with different architectures (S2V, ChebNet), loss functions (L_{MSE}, L_{RANK}) and target transformations (ω, ω_{deg}) and also multiple aspects of the evaluation process: various metrics, baseline models, graph families and we evaluated our models using B&B as well; and each of these degrees of freedom brought us more knowledge about others.

In one such example from Section 3.2.4, we found out that ChebNet model is much more stable than S2V, but it also has problems with accumulating larger values for larger cliques. Since $\omega(v)$ does not have to be correlated with vertex degree, however, a model that can accumulate embeddings only from selected neighbours, such as graph attention network [44], might be a right choice for solving the MCP in the future.

With this approach, we came up with many more ideas for possible future improvements that can be found in conclusion sections in chapters 3 and 4.

Although we showed that GNNs can exploit structural properties of graphs to detect cliques, it is still unclear, what can GNNs actually learn about the nature of cliques and which concrete patterns do trained networks observe in graphs.

A recent line of work showed that capabilities of GNNs are strongly related to Weisfeiler-Lehman algorithm for graph isomorphism [52, 32] and in the future work, an analysis using the perspective of Weisfeiler-Lehman algorithm might lead to theoretical bounds on capabilities of GNNs to detect cliques.

The desire to develop models that can fully grasp the concept of cliques can lead to the design of new models that can capture more structural information. One such idea comes from our observation of the network's inability to process regular graphs which is caused by the symmetry of the initial vertex tags $h_v^{(0)}$. This symmetry might be broken by assigning different, e.g. random, initial values of $h_v^{(0)}$ to different vertices.

The interpretability of neural networks was successfully tackled in the field of image processing, where techniques such as feature visualization and attribution [33] were developed. Future development of similar methods for GNNs is required to achieve a clear understanding of their practical capabilities.

Appendix A – Spectral GNNs

In this appendix we provide a short review of the paper by Bronstein et al. [7], where the development of neural networks based on spectral analysis eventually led to graph neural networks.

The authors first focused on signal processing on graphs, where a function $f : V \to \mathbb{R}$ (a scalar field) describes an input signal on graph vertices and the task is to process or filter this signal to obtain the desired output signal, e.g. class labels for vertices.

The main problem with extension of convolution operation on graphs is that no shift operator $x - x'$ exists in graphs and so the convolution of signal f with filter g cannot be directly extended from its original definition in euclidean spaces:

$$(f \star g)(x) = \int f(x - x') * g(x')dx'$$

.

To overcome this problem, the authors used a property of convolution described by the convolution theorem, which states that convolution in spatial domain can be expressed by a scalar product in spectral domain. This concept can be understood by an analogy with the Fourier transform. To filter a specified frequency defined by g from a signal f in spatial domain, one has to compute the correlation across the whole domain as described by the equation above. In a spectral approach, however, one first computes the spectral representation of the filter and of the signal – the amplitudes of selected frequencies in both the filter and the signal – and then the scalar product of these spectra describes the correlation of signal the with the filter.

In a graph setting, the signal f on vertices has to be first decomposed into a sum of elementary functions from an orthogonal basis. While in Fourier transform, these functions are simple sin or cos functions, one way how to obtain an orthogonal basis for graph signal is to compute a spectral decomposition of the graph's laplacian $L = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ where A is the adjacency matrix of the graph and D is a diagonal matrix of vertex degrees. See Figure 4.3 to gain an intuitive understanding of orthogonal functions on graphs.

If we denote the spectral decomposition of laplacian as $L = \Phi\Lambda\Phi^\top$ where Φ is an orthogonal matrix and Λ is a diagonal matrix consisting of eigenvalues, and if we denote the spectrum (coordinates in orthogonal bases) of filter g as $g' = (g'_1, \dots, g'_n)$,

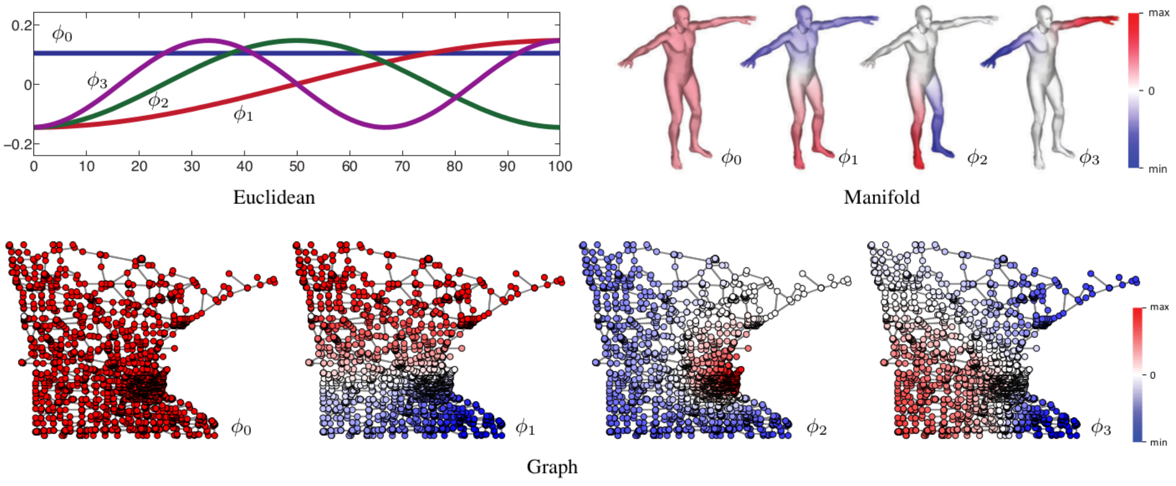

Figure 4.3: Laplacian eigenfunctions $\phi_0, \dots, \phi_3$ on 1D euclidean domain, 2D manifold and on a Minnesota road graph. Reprinted from the original paper [7].

a convolution of signal f with g can be computed as $f \star g = \Phi g' \Phi^\top f$ where f is first transformed to spectral domain with $\Phi^\top$, then multiplied with the filter g' and finally transformed back to spatial domain with Φ.

Similarly as convolution filters in images can detect patterns, such as objects in the image, the graph convolution can detect substructures in graphs. The orthogonality of the basis however leads to periodic base functions and thus the filters might not be localized. To create filters that are focused to a smaller area of the graph, the spectral representation g' has to be smooth and so parametrization of g' with splines was proposed.

Finally to achieve the computational efficiency, when g' is defined using the values of a polynomial expansion in the points of eigenvalues $(\lambda_1, \dots, \lambda_n)$: $g'_i = g'(\lambda_i) = \sum_{j=0}^{n} \alpha_j \lambda_i^j$, the term $\Phi g' \Phi^\top$ can be rewritten as

$$\Phi g'(\Lambda) \Phi^\top = \sum_{j=0}^{n} \alpha_j (\Phi \Lambda \Phi^\top)^j = \sum_{j=0}^{n} \alpha_j L^j$$

and so no eigenvectors and eigenvalues have to be computed [12].

Additionally, the computation of a layer, $h^{(i)} = \sum_{j=0}^{n} \alpha_j L^j h^{(i-1)}$, can be described in a message-passing framework where the polynomial expansion is executed by multiple graph convolutional layers.

Appendix B – Implementation

All code necessary to reproduce the results described in this work can be found at `https://github.com/maaario/mcp-gnns`.

In this repository, we provide the following components of our framework:

- `generator` - A generator of random graphs which serve as training and testing examples for GNNs and are also used for evaluation of B&B.
- `maxclique` - Implementation of branch and bound algorithm. This implementation is used both for for labelling graphs with targets $\omega(v)$ and also for evaluating heuristic functions based on neural networks.
- `clique_finding_models` - Implementation of graph neural networks and the infrastructure used for their training and evaluation.
- `experiments` - Scripts and interactive Python notebooks which produce and visualize the experimental results.

In this appendix we provide a few implementation details and references to libraries we used. More detailed documentation, which contains the installation and usage instructions as well as main architectural features of our components, can be found in `README` files in our repository.

Graph Generator Simple graph-generating procedures are implemented directly in our generator, mostly using NetworkX Python library [18].

To generate smaller instances of DIMACS graphs [21] we used the scripts from the webpage `http://iridia.ulb.ac.be/~fmascia/maximum_clique/` and to generate RB graphs from the BHOSLIB dataset [51] we used a Python script downloaded from `https://github.com/notbad/RB`. These scripts can be automatically downloaded using script `generator/download_third_party_generators.sh`.

B&B We implemented the first version of our branch and bound algorithm according to descriptions in the paper by Tomita et al. [42] and we then generalized this implementation so that various branching and bounding strategies can be used, recursion call metrics can be stored for each run and $\omega(v)$ can be computed for the neighbourhood of each vertex.

In addition to C++ standard libraries, we also used `cxxopts` library from `https://github.com/jarro2783/cxxopts` to parse the commandline arguments.

Neural Networks, Training and Evaluation Framework Our models were implemented using PyTorch [35] and PyTorch Geometric [15] libraries. We used the implementation of Chebyshev convolutional layer from PyTorch Geometric library and we implemented Structure2Vec network using the general messages passing layer using the same library.

To implement the training and evaluation infrastructure, in addition to PyTorch, we mainly used the package PyTorch Ignite (`https://pytorch.org/ignite/`). We are also grateful for the SACRED library [27] which we used to collect and store all the results produced during the training and evaluation of our models.

Appendix C – Target Transform ω_{mc}

Apart from predicting directly $\omega(v)$ or the relative value $\omega_{deg}(v)$ in the regression task, we first considered a transformation of targets, which uses the size of the maximum clique of the graph $\omega(G)$:

$$\omega_{mc}(v) = \frac{\omega(v)}{\omega(G)}$$

This transformation not only maps the targets to $[0, 1]$ interval, but it also uses the whole interval of values, unlike ω_{deg}. Moreover, when experimenting with these targets, our networks achieved unprecedented precision. We illustrate these results with Figure 4.4 and Table 4.1.

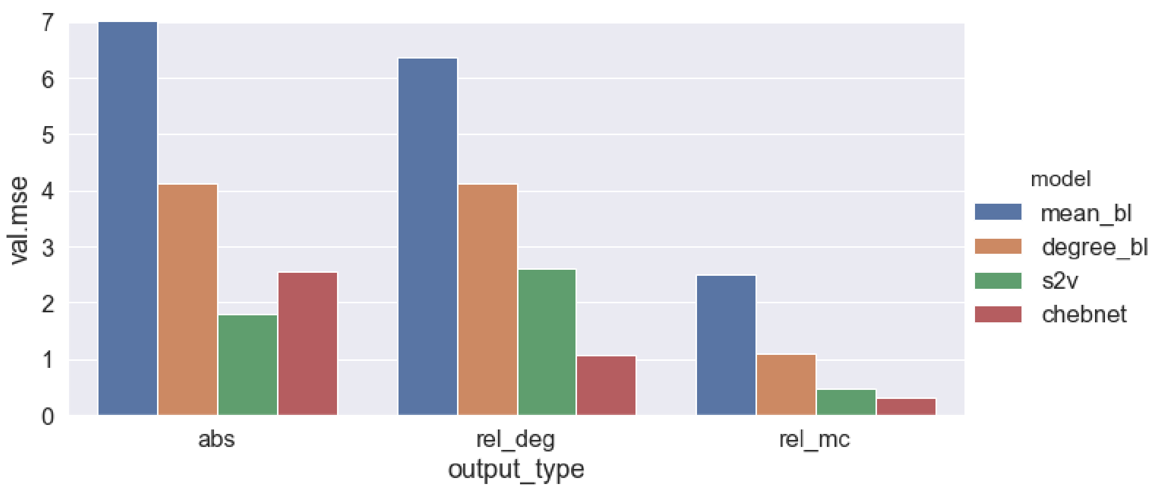

Figure 4.4: Comparison of validation mean square errors for two models (S2V, ChebNet) and two baselines (mean baseline and mean-per-degree baseline) combined with three variants of target transformations ($\omega, \omega_{deg}, \omega_{mc}$), evaluated on testing dataset consisting 100 20-vertex graphs of each type.

A closer look at results revealed that even baseline models perform much better with this transformation and so the success of this method might be mostly caused by transforming the values of $\omega(v)$ for the largest cliques to 1. Since in our graphs, the sizes of most cliques are relatively close to $\omega(G)$, predicting a constant value of approximately 0.95 is quite successful with ω_{mc} targets.

Table 4.1: Performance of ω_{mc} models in regression and ranking tasks evaluated on separate graph types in addition to evaluation on the whole dataset (all).

	MSE				FMC			
	mean	deg	s2v	cheb	mean	deg_den	s2v	cheb
all	2.50	1.09	0.48	**0.32**	.664	.867	.871	**.896**
C.20.5	0.50	0.33	0.32	**0.28**	.497	.754	.763	**.816**
C.20.9	0.28	0.16	0.17	**0.15**	.793	.931	.930	**.946**
brock20	6.88	1.87	0.93	**0.57**	.703	.912	.912	**.924**
dsjc20	3.68	1.54	0.78	**0.42**	.603	.901	.908	**.937**
rb5-4	0.22	**0.17**	0.20	0.18	.724	.838	.841	**.858**

A more serious issue with ω_{mc} is that to predict the absolute $\omega(v)$, we would have to know the value of $\omega(G)$ of the graph, which renders ω_{mc} as impractical for the regression task and thus not useful as a bounding heuristic in B&B. It might still seem useful for the ranking task, however.

The last nail in the coffin for this approach was the observation that with these targets, the network is expected to produce different outputs for the vertices $u \in V, v \in V'$ with equal $\omega(u) = \omega(v)$ if they are contained in graphs G, G' with different clique numbers $\omega(G) \neq \omega(G')$.

Therefore, to predict values of ω_{mc} correctly, apart from $\omega(v)$, each vertex has to be aware of the largest clique of the graph. Since in our architectures all vertices only receive information from vertices in distance at most t, where t is the number of convolutional layers, there is no global information flow in larger or sparser graphs in the network, and so this requirement makes the task practically impossible for our networks. Providing the network with the global information flow (e.g. in a form of central node) just because of this reason seems as a very complicated solution in comparison to changing the definition of targets.

Although ω_{mc} might be usable for smaller or dense graphs where the value of $\omega(G)$ can be expected beforehand (e.g. in random graphs with a fixed edge probability), we decided not to study this variant further.

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[2] Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov, and Panos M. Pardalos. Improvements to mcs algorithm for the maximum clique problem. *Journal of Combinatorial Optimization*, 27(2):397–416, 2014.

[3] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *CoRR*, abs/1811.06128, 2018.

[6] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604. Springer International Publishing, 2018.

[7] Michael Bronstein, Joan Bruna, Yann Lecun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34, 2016.

[8] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2013.

[9] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2017.

[10] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2702–2711. JMLR.org, 2016.

[11] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *CoRR*, abs/1704.01665, 2017.

[12] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems 29*, pages 3844–3852. Curran Associates, Inc., 2016.

[13] Brendan L Douglas. The weisfeiler-lehman method and graph isomorphism testing. *arXiv preprint arXiv:1101.5211*, 2011.

[14] Torsten Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In Rolf Möhring and Rajeev Raman, editors, *Algorithms — ESA 2002*, pages 485–498. Springer Berlin Heidelberg, 2002.

[15] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[16] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 1263–1272. JMLR.org, 2017.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[18] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[19] Johan Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182(1):105–142, 1999.

[20] Arun Jagota, Laura A. Sanchis, and Ravikanth Ganesan. Approximately solving maximum clique using neural network and related heuristics. In *Cliques, Coloring, and Satisfiability*, 1993.

[21] David. J. Johnson and Michael. A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993.* American Mathematical Society, Boston, MA, USA, 1996.

[22] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39:378–406, 1991.

[23] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer US, 1972.

[24] Elias B. Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 724–731. AAAI Press, 2016.

[25] Elias Boutros Khalil, Bistra N. Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *IJCAI*, 2017.

[26] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[27] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49 – 56, 2017.

[28] Janez Konc and Dušanka Janežič. An improved branch and bound algorithm for the maximum clique problem. In *MATCH - Communications in Mathematical and in Computer Chemistry*, volume 58, pages 569–590, 2007.

[29] Deniss Kumlander. A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search. *Proceedings of the Fourth International Conference on Engineering Computational Technology*, pages 202–208, 2004.

[30] Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 128–133, 2010.

[31] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *CoRR*, abs/1810.10659, 2018.

[32] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. *CoRR*, abs/1810.02244, 2018.

[33] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. https://distill.pub/2018/building-blocks.

[34] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Neural Information Processing Systems - Workshop*, 2017.

[36] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[37] Farah Sarwar and Abdul Aziz Bhatti. Critical analysis of hopfield's neural network model for tsp and its comparison with heuristic algorithm for shortest path computation. In *Proceedings of 2012 9th International Bhurban Conference on Applied Sciences Technology (IBCAST)*, pages 111–114, 2012.

[38] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[39] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[41] Etsuji Tomita and Toshikatsu Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, 37(1):95–111, 2007.

[42] Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete Mathematics and Theoretical Computer Science*, pages 278–289. Springer Berlin Heidelberg, 2003.

[43] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation*, pages 191–203, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[44] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[45] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015.

[46] Rong Long Wang, Zheng Tang, and Qi Ping Cao. An efficient approximation algorithm for finding a maximum clique using hopfield network learning. *Neural Computation*, 15(7):1605–1619, 2003.

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

[48] Wikipedia. Evaluation measures (information retrieval). In Wikipedia, the free encyclopedia, 2019. Retrieved May 1, 2019, from `https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)`.

[49] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242:693–709, 2015.

[50] Ke Xu, Frederic Boussemart, Fred Hemery, and Christophe Lecoutre. A simple model to generate hard satisfiable instances. *IJCAI International Joint Conference on Artificial Intelligence*, 2005.

[51] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171:514–534, 2007.

[52] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018.