WHAT'S WRONG WITH DEEP LEARNING IN TREE SEARCH FOR COMBINATORIAL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Combinatorial optimization lies at the core of many real-world problems. Especially since the rise of graph neural networks (GNNs), the deep learning community has been developing solvers that derive solutions to NP-hard problems by learning the problem-specific solution structure. However, reproducing the results of these publications proves to be difficult. We make three contributions. First, we present an open-source benchmark suite for the NP-hard MAXIMUM INDEPENDENT SET problem, in both its weighted and unweighted variants. The suite offers a unified interface to various state-of-the-art traditional and machine learning-based solvers. Second, using our benchmark suite, we conduct an in-depth analysis of the popular guided tree search algorithm by Li et al. [NeurIPS 2018], testing various configurations on small and large synthetic and real-world graphs. By re-implementing their algorithm with a focus on code quality and extensibility, we show that the graph convolution network used in the tree search does not learn a meaningful representation of the solution structure, and can in fact be replaced by random values. Instead, the tree search relies on algorithmic techniques like graph kernelization to find good solutions. Thus, the results from the original publication are not reproducible. Third, we extend the analysis to compare the tree search implementations to other solvers, showing that the classical algorithmic solvers often are faster, while providing solutions of similar quality. Additionally, we analyze a recent solver based on reinforcement learning and observe that for this solver, the GNN is responsible for the competitive solution quality.

1 INTRODUCTION

Various communities have been dealing with the question of how to efficiently solve combinatorial problems, which frequently are NP-hard. These problems often have real-world applications in industry, for instance in staff assignment (Peters et al., 2019), supply chain optimization (Eskandarpour et al., 2015), and traffic optimization (Böther et al., 2021).

In recent years, also the machine learning community has been engaged in solving combinatorial problems. The reason why machine learning might be able to solve such problems faster compared to classical solvers like Gurobi (Gurobi Optimization LLC, 2021) is that the solutions to a specific problem often follow a certain structure, which we might be able to learn given enough data. Commonly used machine learning approaches for combinatorial optimization include Deep Reinforcement Learning (Khalil et al., 2017; Ahn et al., 2020) and Graph Neural Networks (GNNs) (Cappart et al., 2021; Li et al., 2018).

However, these statistics-based techniques have the well-known problem of reproducibility (Ioannidis, 2005; Baker, 2016), which is a particular problem in machine learning research with untested or even unpublished code and data sets (Kapoor & Narayanan, 2020; Ding et al., 2020), and is the reason for the interest in initiatives such as Papers With Code[1] and the ReScience Journal[2]. Keeping the importance of reproducibility in mind, we evaluate various machine learning approaches for combinatorial optimization, and compare them to traditional solvers. Following previous research (Li et al., 2018; Ahn et al., 2020), we focus on the MAXIMUM (WEIGHTED) INDEPENDENT SET problem (Miller & Muller, 1960; Karp, 1972). Overall, we contribute the following.

[1] https://paperswithcode.com/
[2] https://rescience.github.io/

- We provide an open-source, extensible benchmark suite for MAXIMUM (WEIGHTED) INDEPENDENT SET solvers. The software currently supports five state-of-the-art solvers, INTEL-TREESEARCH (Li, Chen, and Koltun, 2018), GUROBI (Gurobi Optimization LLC, 2021), KAMIS (Lamm et al., 2017; Hespe et al., 2019; Lamm et al., 2019), LEARNING WHAT TO DEFER (Ahn, Seo, and Shin, 2020), and DGL-TREESEARCH. Our DGL-TREESEARCH is a modern re-implementation of the INTEL-TREESEARCH, implemented in PyTorch (Paszke et al., 2019) and the Deep Graph Library (Wang et al., 2019), with a focus on clean, readable code, as well as performance, and it fixes various issues of the original code. Our evaluation suite lays the ground for further research on hard combinatorial (graph) problems and aims at providing a fair and comparable environment for further evaluations.
- Using our re-implementation of the tree search, we propose and analyze additional techniques aiming at improving the guided search. Employing the benchmark suite, we conduct an exhaustive analysis of various configurations of the tree search algorithms, showing that the results of the highly-cited INTEL-TREESEARCH approach are not reproducible, neither with the original code nor with our re-implementation. When exploring the design space further, we show that the various techniques used by the tree search algorithm to improve the results, like graph kernelization, are the reason for good performance, especially on hard data sets. In fact, replacing the GNN output with random values performs similar to using the trained network.
- Having analyzed the configuration space, we compare the tree search approaches to the classical solvers like GUROBI and KAMIS, showing that problem-tailored solvers are often the superior approach. Without using techniques like graph reduction in the tree search, classical solvers are superior. The classical solvers show to be more efficient even when accessing these routines – that have been implemented for the algorithmic solvers in the first place – in the tree search. Last, we show that LEARNING WHAT TO DEFER seems to be able to find good results very quickly, indicating that unsupervised reinforcement learning for combinatorial problems is a promising direction for future research.

The remainder of this paper is structured as follows. In Section 2, we introduce the various solvers and how they are integrated into our open-source benchmark suite, before evaluating them in Section 3. We discuss some related work in Section 4, and conclude in Section 5.

2 INDEPENDENT SET SOLVERS

In this section, we formally introduce the MAXIMUM INDEPENDENT SET (MIS) problem as well as the solvers included in our analysis.

Given an undirected graph $G = (V, E)$, an independent set is a set of vertices $S \subseteq V$ for which for all vertices $u, v \in S$, $(u, v) \notin E$. For $u \in V$, let w_u be its weight, and let $\text{IS}(G)$ be all independent sets of G, then the MAXIMUM WEIGHTED INDEPENDENT SET (MWIS) problem aims at determining

$$\arg\max_{S \in \text{IS}(G)} \sum_{u \in S} w_u.$$

The unweighted MIS problem is equivalent to the MWIS problem, where f.a. $u \in V$, $w_u = 1$. Both problems are strongly NP-complete (Garey & Johnson, 1978). Next, we briefly explain the solvers that we use to find such maximum independent sets.

Gurobi. GUROBI is a commercial mathematical optimization solver. We access GUROBI in Python using the abstraction layer PuLP[3], and formulate the MWIS problem as a linear program.

KaMIS. KAMIS is an open-source solver tailored towards the MIS and MWIS problems. It offers support both for the unweighted case (Lamm et al., 2017; Hespe et al., 2019) as well as the weighted case (Lamm et al., 2019). It employs graph kernelization and an optimized branch-and-bound algorithm to efficiently find independent sets. Note that the algorithms and techniques differ between the weighted and unweighted cases. We use the code unmodified from the official repository[4].

[3] `https://coin-or.github.io/pulp/`
[4] `https://github.com/KarlsruheMIS/KaMIS`

Intel-TreeSearch. In their influential paper, Li et al. (2018) propose a guided tree search algorithm to find maximum independent sets of a graph. The idea is to train a graph convolutional network (GCN) (Kipf & Welling, 2017), which assigns each vertex a probability of belonging to the independent set, and then greedily and iteratively assign vertices to the set. They furthermore employ the reduction and local search algorithms by KAMIS to speed up the computation. We use their published code[5], which unfortunately is not runnable in its default state. We apply a git patch[6] to make the code runnable, enable further evaluation by collecting statistics, and add command-line flags for more fine-grained control of the solver configuration. For a detailed explanation of the tree search algorithm and details of the model, we refer to Appendix B as well as the original paper. In Appendix C, we give some details on possible complications with the original code.

DGL-TreeSearch. Because the code provided by Li et al. (2018) might be difficult to read and maintain, and hence is prone to errors in the evaluation, we re-implement the tree search using PyTorch (Paszke et al., 2019) and the established Deep Graph Library (Wang et al., 2019). Our implementation aims at offering a more readable and modern implementation, which benefits from improvements in the two deep learning libraries during recent years. Furthermore, it fixes various issues of the original implementation that sometimes deviates from the paper. Additionally, we implement further techniques to improve the search, like queue pruning, and weighted selection of the next element, as well as multi-GPU functionality. The code is delivered in conjunction with the benchmarking suite.

Learning What To Defer. We test LEARNING WHAT TO DEFER (LWD), an unsupervised deep reinforcement learning-based solution introduced by Ahn et al. (2020). Their idea is similar to the tree search, as the algorithm iteratively assigns vertices to the independent set. However, this is not done using a supervised GCN, but instead by an unsupervised agent built upon the GraphSAGE architecture (Hamilton et al., 2017) and trained by Proximal Policy Optimization (Schulman et al., 2017). There is no queue of partial solutions. We refer to the original paper for details on the algorithm. As their code[7] does not work with generic input, we patch their code.

Our open-source benchmarking suite[8] integrates all these solvers in one easily accessible command-line interface using Anaconda (Anaconda Inc., 2020), with a unified input and output format. We provide our code for DGL-TREESEARCH and our GUROBI interface directly, and download, compile, and patch the other solvers on-demand. It handles the correct invocation of the solvers and allows to quickly run experiments on various solvers in different configurations.

3 EVALUATION

In this section, we first introduce our experimental setup and then focus on the analysis of the supervised tree search approach for combinatorial optimization in Section 3.1. After having derived a good configuration for the tree search algorithm, we continue to compare the tree search algorithms to the other classical and reinforcement learning solvers in Section 3.2. We investigate the scalability of the approaches in Section 3.3, and analyze the behavior on the weighted MIS problem in Section 3.4.

Experimental Setup. We run all our experiments on an NVIDIA DGX-1 with two 20-Core Intel Xeon E5-2698 CPUs at 2.2 GHz, leading to overall 40 physical and 80 logical cores, 512 GB of memory, and a total of eight NVIDIA Tesla V100 GPUs. For each experiment, we explicitly state the number of threads and GPUs used. We run Ubuntu 20.04.1 LTS, using Linux kernel `4.15.0-124`.

Datasets. We evaluate the various solvers and configurations using both real-world datasets as well as generated random graphs in various sizes. We make use of the random graph models by Erdős & Rényi (1960) (ER), Albert & Barabási (2002) (BA), Holme & Kim (2002) (HK), Watts & Strogatz (1998) (WS) and the Hyperbolic Random Graph model (Krioukov et al., 2010) (HRG). The graph

[5] `https://github.com/isl-org/NPHard`

[6] A git patch is a file transparently stating the changes we require to make the code compatible with our benchmarking suite.

[7] `https://github.com/sungsoo-ahn/learning_what_to_defer`

[8] Anonymous repository: `https://anonymous.4open.science/r/mwis-benchmark`

Table 1: Results of the tree searches in various configurations. We run experiments on both synthetic and real-world graphs with varying numbers of nodes (c.f. Appendix D). For all configurations, in the first row, we state the average MIS size as well as the average approximation factor for graphs where we were able to pre-calculate the provably optimal MIS using GUROBI. In the second row, the average time in seconds until the best solution was found, and, in brackets, the number of graphs where any solution was found are given. The average values refer only to the graphs within a dataset for which a solution was found. Columns that start with a plus (+) add an additional flag to the previous column; columns that do not start with a plus use *just* the stated flags. For Intel, we start with the default setting (`d`), proceed to include the graph reduction (`+r`), and then the local search (`+ls`), and last test reduction and local search in a multithreaded setup (`+mt`). For DGL, we explore the configuration space further. After analyzing the default (`d`), adding reduction (`+r`), and local search (`+ls`), we test configurations replacing the GCN with random outputs (`rand`), using queue pruning (`qp`), and weighted queue pop (`+wp`). We also test a combination of reduction, queue pruning, and weighted queue pop (`+r`), proceed to add the local search (`+ls`), and again replace the GCN by randomly generated outputs (`+rand`). The full configuration is tested with and without random output in a multithreaded setup (`+mt`). All multithreaded experiments in this table use 8 threads, and all threads share a single GPU. For the random graphs of size 50-100, all solvers have a time limit of 15 seconds; for the SATLIB, as-caida, PPI, REDDIT, Citeseer, Cora, and ego-facebook datasets the time limit is 30 seconds; for the bitcoin, PubMed, Wikipedia, VC-BM and roadnet-berlin graphs, the time limit is 5 minutes. We refer to Appendix B for detailed explanations of the techniques used. We mark the configuration/solver that has the best average MIS in bold, and grey out configurations for which solutions were found for less than 20 % of all graphs.

		Intel				DGL										
Graph	Nodes	d	+r	+ls	+mt	d	+r	+ls	rand	qp	+wp	+r	+ls	+rand	r+qp+wp+ls+mt	+rand
ER	50-100	20.58 (0.98)	20.76 (0.99)	**20.83 (1.00)**	**20.83 (1.00)**	19.88 (0.95)	20.58 (0.98)	**20.83 (1.00)**	19.05 (0.91)	19.69 (0.94)	19.16 (0.92)	20.54 (0.98)	**20.83 (1.00)**	**20.83 (1.00)**	**20.83 (0.99)**	20.82 (0.99)
		1.70 (500)	0.81 (500)	0.02 (500)	5.39 (500)	3.53 (500)	1.89 (500)	0.28 (500)	5.26 (500)	2.66 (500)	3.39 (500)	1.59 (500)	0.26 (500)	0.31 (500)	0.41 (500)	0.37 (499)
	700-800	39.90 (-)	39.71 (-)	44.08 (-)	**44.98 (-)**	37.13 (-)	38.51 (-)	43.90 (-)	33.69 (-)	37.06 (-)	34.79 (-)	38.31 (-)	43.91 (-)	44.02 (-)	44.32 (-)	44.55 (-)
		12.00 (100)	19.73 (98)	16.81 (100)	11.90 (100)	13.63 (100)	15.91 (100)	12.82 (100)	15.71 (100)	14.76 (100)	9.11 (100)	12.52 (100)	9.30 (100)	10.08 (100)	22.97 (96)	17.95 (99)
BA	50-100	42.42 (0.99)	**42.43 (1.00)**	**42.43 (1.00)**	**42.43 (1.00)**	42.05 (0.99)	**42.43 (1.00)**	**42.43 (1.00)**	40.81 (0.96)	41.72 (0.98)	41.74 (0.98)	**42.43 (1.00)**	**42.43 (1.00)**	**42.43 (1.00)**	**42.43 (1.00)**	**42.43 (1.00)**
		0.54 (500)	0.00 (500)	0.00 (500)	4.97 (500)	2.44 (500)	0.08 (500)	0.08 (500)	5.52 (500)	1.84 (500)	1.64 (500)	0.07 (500)	0.08 (500)	0.12 (500)	0.11 (500)	0.12 (500)
	700-800	420.89 (0.97)	**432.58 (1.00)**	**432.58 (1.00)**	**432.58 (1.00)**	389.65 (0.90)	**432.58 (1.00)**	**432.58 (1.00)**	374.37 (0.86)	398.90 (0.92)	406.82 (0.94)	**432.58 (1.00)**	**432.58 (1.00)**	**432.58 (1.00)**	**432.58 (1.00)**	**432.58 (1.00)**
		3.09 (18)	0.01 (100)	0.02 (100)	5.31 (100)	16.50 (80)	0.45 (100)	0.50 (100)	20.08 (19)	19.09 (86)	6.92 (100)	0.39 (100)	0.43 (100)	0.67 (100)	0.69 (100)	0.73 (100)
HK	50-100	42.32 (0.99)	**42.33 (1.00)**	**42.33 (1.00)**	**42.33 (1.00)**	41.87 (0.98)	**42.33 (1.00)**	**42.33 (1.00)**	40.87 (0.96)	41.50 (0.98)	41.45 (0.98)	**42.33 (1.00)**	**42.33 (1.00)**	**42.33 (1.00)**	**42.33 (1.00)**	**42.33 (1.00)**
		0.53 (500)	0.00 (500)	0.00 (500)	0.00 (500)	2.59 (500)	0.06 (500)	0.07 (500)	5.78 (500)	2.08 (500)	1.77 (500)	0.07 (500)	0.07 (500)	0.08 (500)	0.07 (500)	0.08 (500)
	700-800	417.89 (0.98)	**429.81 (1.00)**	**429.81 (1.00)**	**429.81 (1.00)**	390.64 (0.90)	**429.81 (1.00)**	**429.81 (1.00)**	370.05 (0.86)	393.10 (0.91)	407.04 (0.94)	**429.81 (1.00)**	**429.81 (1.00)**	**429.81 (1.00)**	**429.81 (1.00)**	**429.81 (1.00)**
		4.23 (19)	0.01 (100)	0.01 (100)	5.26 (100)	19.66 (83)	0.37 (100)	0.38 (100)	22.29 (19)	18.15 (80)	7.50 (100)	0.42 (100)	0.39 (100)	0.44 (100)	0.42 (100)	0.43 (100)
WS	50-100	38.69 (0.99)	**38.70 (1.00)**	**38.70 (1.00)**	**38.70 (1.00)**	37.91 (0.97)	**38.70 (0.99)**	**38.70 (1.00)**	36.58 (0.94)	37.60 (0.97)	36.94 (0.95)	**38.70 (0.99)**	**38.70 (1.00)**	**38.70 (1.00)**	**38.70 (1.00)**	**38.70 (1.00)**
		0.53 (500)	0.00 (500)	0.00 (500)	0.00 (500)	3.80 (500)	0.07 (500)	0.07 (500)	5.22 (500)	2.42 (500)	2.17 (500)	0.08 (500)	0.08 (500)	0.08 (500)	0.07 (500)	0.08 (500)
	700-800	- (-)	**386.90 (1.00)**	**386.90 (1.00)**	**386.90 (1.00)**	- (-)	**386.90 (1.00)**	**386.90 (1.00)**	326.71 (0.87)	- (-)	368.64 (0.95)	**386.90 (1.00)**	**386.90 (1.00)**	**386.90 (1.00)**	**386.90 (1.00)**	**386.90 (1.00)**
		-	0.00 (100)	0.00 (100)	5.49 (100)	-	0.37 (100)	0.36 (100)	21.34 (7)	-	13.02 (87)	0.40 (100)	0.42 (100)	0.43 (100)	0.39 (100)	0.43 (100)
HRG	50-100	33.62 (0.99)	**33.72 (1.00)**	**33.72 (1.00)**	**33.72 (1.00)**	32.80 (0.97)	**33.72 (1.00)**	**33.72 (1.00)**	32.40 (0.96)	32.62 (0.96)	30.93 (0.92)	**33.72 (1.00)**	**33.72 (1.00)**	**33.72 (1.00)**	**33.72 (1.00)**	**33.72 (1.00)**
		3.24 (495)	0.00 (500)	0.00 (500)	0.00 (500)	5.50 (500)	0.07 (500)	0.06 (500)	5.14 (500)	4.23 (500)	3.45 (500)	0.06 (500)	0.07 (500)	0.07 (500)	0.06 (500)	0.07 (500)
	700-800	- (-)	**304.21 (1.00)**	**304.21 (1.00)**	**304.21 (1.00)**	221.80 (0.75)	**304.21 (1.00)**	**304.21 (1.00)**	260.27 (0.86)	221.67 (0.74)	245.29 (0.80)	**304.21 (1.00)**	**304.21 (1.00)**	**304.21 (1.00)**	**304.21 (1.00)**	**304.21 (1.00)**
		-	0.01 (100)	0.01 (100)	5.51 (100)	17.81 (5)	0.35 (100)	0.40 (100)	20.57 (15)	21.23 (6)	13.55 (98)	0.39 (100)	0.39 (100)	0.44 (100)	0.44 (100)	0.42 (100)
SATLIB	1209-1 347	- (-)	424.86 (0.99)	426.39 (0.99)	426.51 (0.99)	340.50 (0.79)	421.93 (0.98)	426.25 (0.99)	- (-)	- (-)	356.93 (0.84)	419.61 (0.98)	426.37 (0.99)	426.48 (0.99)	426.55 (0.99)	**426.57 (0.99)**
		-	11.54 (500)	6.89 (500)	7.99 (500)	18.48 (2)	16.42 (500)	9.55 (500)	-	-	17.91 (211)	12.57 (500)	6.23 (500)	5.58 (500)	13.12 (500)	10.34 (500)
VC-BM	450-1 534	39.85 (0.90)	39.08 (0.91)	44.26 (0.99)	**45.10 (1.00)**	37.20 (0.85)	38.70 (0.90)	44.35 (0.99)	35.45 (0.82)	37.70 (0.85)	35.42 (0.78)	38.38 (0.86)	44.52 (0.99)	44.58 (0.99)	44.65 (0.99)	44.88 (0.99)
		149.80 (40)	169.49 (38)	166.04 (39)	51.64 (40)	139.55 (40)	138.83 (40)	84.08 (40)	123.72 (40)	119.78 (40)	68.33 (40)	93.88 (40)	73.67 (40)	64.57 (40)	145.24 (40)	113.43 (40)
as-Caida	8020-26 475	- (-)	19387.47 (0.99)	19387.47 (0.99)	19387.47 (0.99)	- (-)	**19388.03 (1.00)**	19387.90 (0.99)	- (-)	- (-)	- (-)	**19388.03 (1.00)**	19387.90 (0.99)	19387.90 (0.99)	19387.90 (0.99)	19387.90 (0.99)
		-	0.26 (122)	8.41 (122)	14.49 (122)	-	7.32 (122)	11.22 (122)	-	-	-	7.62 (122)	12.27 (122)	14.48 (122)	15.63 (122)	12.74 (122)
PPI	591-3 480	- (-)	1002.12 (0.99)	**1002.83 (1.00)**	1002.71 (0.99)	269.00 (0.97)	1001.83 (0.99)	1002.67 (0.99)	- (-)	372.67 (0.84)	804.38 (0.92)	1001.67 (0.99)	1002.67 (0.99)	1002.79 (0.99)	1002.50 (0.99)	1002.58 (0.99)
		-	14.59 (24)	24.70 (24)	8.73 (24)	23.56 (1)	7.14 (24)	3.70 (24)	-	19.86 (3)	20.11 (13)	9.81 (24)	3.68 (24)	4.11 (24)	3.70 (24)	4.82 (24)
REDDIT-B	46-2 283	146.06 (0.99)	**287.54 (1.00)**	**287.54 (1.00)**	**287.54 (1.00)**	186.31 (0.87)	**287.54 (1.00)**	**287.54 (1.00)**	248.96 (0.96)	198.68 (0.89)	249.27 (0.92)	**287.54 (1.00)**	**287.54 (1.00)**	**287.54 (1.00)**	**287.54 (1.00)**	**287.54 (1.00)**
		5.94 (200)	0.00 (500)	0.02 (500)	5.37 (500)	11.81 (406)	0.18 (500)	0.20 (500)	15.23 (475)	12.29 (424)	4.93 (492)	0.20 (500)	0.24 (500)	0.26 (500)	0.26 (500)	0.27 (500)
REDDIT-5K	55-3 648	194.27 (0.99)	586.31 (0.99)	586.31 (0.99)	586.31 (0.99)	252.74 (0.88)	**586.32 (1.00)**	**586.32 (1.00)**	378.83 (0.94)	255.66 (0.89)	416.49 (0.92)	**586.32 (1.00)**	**586.32 (1.00)**	**586.32 (1.00)**	**586.32 (1.00)**	**586.32 (1.00)**
		7.97 (37)	0.01 (500)	0.04 (500)	5.35 (500)	15.36 (152)	0.33 (500)	0.37 (500)	19.41 (325)	17.87 (168)	9.63 (396)	0.37 (500)	0.46 (500)	0.44 (500)	0.48 (500)	0.50 (500)
REDDIT-12K	41-897	128.61 (0.99)	**170.20 (1.00)**	**170.20 (1.00)**	**170.20 (1.00)**	136.05 (0.92)	**170.20 (1.00)**	**170.20 (1.00)**	163.15 (0.98)	145.23 (0.95)	155.81 (0.93)	**170.20 (1.00)**	**170.20 (1.00)**	**170.20 (1.00)**	**170.20 (1.00)**	**170.20 (1.00)**
		3.99 (349)	0.00 (500)	0.01 (500)	5.20 (500)	9.53 (467)	0.12 (500)	0.14 (500)	11.46 (497)	8.17 (478)	3.86 (500)	0.14 (500)	0.16 (500)	0.16 (500)	0.17 (500)	0.17 (500)
wiki-RfA	11 380	- (-)	**8111.00 (1.00)**	**8111.00 (1.00)**	**8111.00 (1.00)**	- (-)	8096.00 (0.99)	8096.00 (0.99)	- (-)	- (-)	8072.00 (0.99)	8096.00 (0.99)	8096.00 (0.99)	8096.00 (0.99)	8096.00 (0.99)	8096.00 (0.99)
		-	0.86	6.50	13.08	-	10.79	16.14	-	-	270.94	12.41	19.12	20.40	19.95	26.38
wiki-Vote	7 115	- (-)	**4866.00 (1.00)**	**4866.00 (1.00)**	**4866.00 (1.00)**	- (-)	4864.00 (0.99)	4864.00 (0.99)	- (-)	- (-)	4748.00 (0.97)	4864.00 (0.99)	4864.00 (0.99)	4864.00 (0.99)	4864.00 (0.99)	4864.00 (0.99)
		-	0.45	2.60	8.56	-	7.79	12.05	-	-	236.68	10.82	13.99	19.11	11.20	24.41
PubMed	19 717	- (-)	**15912.00 (1.00)**	**15912.00 (1.00)**	**15912.00 (1.00)**	- (-)	15888.00 (0.99)	15888.00 (0.99)	- (-)	- (-)	- (-)	15888.00 (0.99)	15888.00 (0.99)	15888.00 (0.99)	15888.00 (0.99)	15888.00 (0.99)
		-	0.25	6.07	12.91	-	12.32	17.12	-	-	-	14.77	18.44	20.44	23.34	29.13
Cora	2 708	- (-)	**1451.00 (1.00)**	**1451.00 (1.00)**	**1451.00 (1.00)**	- (-)	**1451.00 (1.00)**	**1451.00 (1.00)**	- (-)	- (-)	- (-)	**1451.00 (1.00)**	**1451.00 (1.00)**	**1451.00 (1.00)**	**1451.00 (1.00)**	**1451.00 (1.00)**
		-	0.03	0.21	5.63	-	7.88	6.27	-	-	-	9.73	11.48	21.09	13.63	6.38
Citeseer	3 264	- (-)	**1808.00 (1.00)**	**1808.00 (1.00)**	**1808.00 (1.00)**	- (-)	**1808.00 (1.00)**	**1808.00 (1.00)**	- (-)	- (-)	- (-)	**1808.00 (1.00)**	**1808.00 (1.00)**	**1808.00 (1.00)**	**1808.00 (1.00)**	**1808.00 (1.00)**
		-	0.03	0.24	5.51	-	9.21	6.19	-	-	-	9.60	11.96	17.24	14.62	7.34
bitcoin-alpha	3 783	- (-)	**2718.00 (1.00)**	**2718.00 (1.00)**	**2718.00 (1.00)**	- (-)	2716.00 (0.99)	2716.00 (0.99)	- (-)	- (-)	2658.00 (0.97)	2716.00 (0.99)	2716.00 (0.99)	2716.00 (0.99)	2716.00 (0.99)	2716.00 (0.99)
		-	0.05	0.50	6.10	-	6.65	5.28	-	-	76.54	8.58	8.91	12.33	11.41	16.35
bitcoin-otc	5 881	- (-)	4346.00 (0.99)	4346.00 (0.99)	4346.00 (0.99)	- (-)	**4352.00 (1.00)**	**4352.00 (1.00)**	- (-)	- (-)	- (-)	**4352.00 (1.00)**	**4352.00 (1.00)**	**4352.00 (1.00)**	**4352.00 (1.00)**	**4352.00 (1.00)**
		-	0.08	1.13	6.86	-	7.44	6.19	-	-	-	10.33	10.35	12.95	12.47	14.57
ego-facebook	4 039	- (-)	- (-)	- (-)	**1046.00 (1.00)**	- (-)	1030.00 (0.98)	**1046.00 (1.00)**	- (-)	- (-)	- (-)	1030.00 (0.98)	**1046.00 (1.00)**	**1046.00 (1.00)**	- (-)	**1046.00 (1.00)**
		-	-	-	14.34	-	17.04	11.82	-	-	-	29.87	14.94	23.95	-	14.12
roadnet-berlin	61 204	- (-)	- (-)	- (-)	29843.00 (0.99)	- (-)	29792.00 (0.99)	29792.00 (0.99)	- (-)	- (-)	- (-)	29792.00 (0.99)	29792.00 (0.99)	29792.00 (0.99)	**29888.00 (1.00)**	**29888.00 (1.00)**
		-	-	-	144.22	-	28.27	34.13	-	-	-	33.04	39.53	39.21	68.22	52.70

generation functionality, employing NetworkX (Hagberg et al., 2008) and girgs (Bläsius et al., 2019) as backends, is integrated into our benchmark suite. For real-world datasets, we focus on the SATLIB dataset (Hoos & Stützle, 2000), which consists of synthetic 3-SAT instances, and the vertex cover benchmark (VC-BM) (Xu et al., 2007) consisting of 40 graphs on which finding the maximum independent set is synthetically made hard; we also test various other graphs, like citation networks. Details on these datasets, as well as detailed descriptions and splits of all mentioned datasets, and hyperparameters of graph generation, can be found in Appendix D.

3.1 Analysis of the Tree Search

In this subsection, we analyze the Intel-TreeSearch and DGL-TreeSearch. These supervised approaches require training a GCN (Kipf & Welling, 2017). Following Li et al. (2018), we train both on the SATLIB dataset for 20 epochs using the Adam optimizer (Kingma & Ba, 2015). As the GCN outputs multiple probability maps (c.f. Appendix B), we employ the hindsight loss, which, for multiple choices, outputs the loss of the best choice (Guzmán-Rivera et al., 2012; Chen & Koltun, 2017). We fix the number of probability maps to 32, to enable comparison with Li et al. (2018). We test the solvers in different configurations on various graphs; these configurations are assorted with increasing levels of complexity and intuitively should improve the solution quality. Details on what effects the individual configuration options have are given in Appendix B. The results can be found in Table 1.

First, we discuss the results of the random graphs. For all small graphs, both tree searches in all configurations find solutions most of the time. These are close to optimal as soon as reduction (`+r`) and local search (`+ls`) are involved. Notably, on larger graphs, in the default variants, where neither reduction nor local search is enabled, both Intel-TreeSearch and DGL-TreeSearch do not often find a solution. This shows that for medium-sized graphs, vanilla tree searches cannot discover solutions within a feasible time limit. The reductions often single-handedly solve the problem instance, as seen for example on the simple BA graphs, where a solution is found instantaneously.

To approach the issue of requiring hand-tailored reduction techniques to obtain any solution, with DGL-TreeSearch, we analyze queue pruning and the weighted queue pop. While queue pruning itself performs similar to the default configuration for all graphs, the weighted pop vastly increases the number of solutions found. For example, for large hyperbolic random graphs, in the default configuration, the DGL-TreeSearch was only able to find solutions for 5 graphs (and the Intel-TreeSearch was not able to find any solution), whereas using queue pruning together with the weighted queue pop, we find solutions for 98 % of all large HRGs. A similar observation can be made for WS graphs. Interestingly, it seems like the MIS problem is harder to address with the default approach on graphs that try to model real-world networks, such as the just mentioned HRGs and WS graphs (Bläsius et al., 2018). Overall, queue pruning might be desirable to reduce memory consumption, but its impact on solving quality and time is limited; on the other hand, weighted queue popping is a general idea that brings some more depth-search-like behavior into the breadth-search-like approach at hand, and thus enables the solver to find a lot more solutions.

For small and large BA, HK, WS graphs as well as HRGs, the reduction itself already leads to an achieved average approximation of 1. Only for the large ER graphs, for which Gurobi was not able to find provably optimal assignments, the local search further improves the average independent set size. Multithreading cannot further improve the results of the random graphs.

Now, we discuss the results for real-world graphs. Unlike previously, where we just aimed at finding an MIS on random graphs, solving the MIS problem on SATLIB is equivalent to finding a satisfiable assignment to synthesized hard 3-SAT instances. Thus, one can expect this data set to be particularly hard. Our experiments confirm this, as both implementations rarely find solutions within the time limit without reduction enabled. We can see that, similar to the random graphs, the weighted queue pop enables the search to at least find *some* solution, as this configuration is at least able to find 211 instead of no results at all. However, the average independent set size is 357, which is not very good, considering each MIS in this dataset contains at least 403 vertices.

Considering the other hard dataset, VC-BM, due to the higher time limit, the default configurations are already able to find some solution. Interestingly, for VC-BM, the reductions do not improve the performance of the solvers. Hence, the weighted queue pop is very important for finding results quickly, shrinking the time needed from the default 140 seconds to under 70 seconds. We also see that the weighted pop enables discovering a solution for the Wikipedia datasets and the bitcoin-alpha graph.

These results are interesting for various reasons. First, the models are trained on the SATLIB data, so they do not have to generalize to a different solution structure for this dataset. Still, the default configuration does not find any independent set for this dataset. This contradicts the original paper by Li et al. (2018), which claims an average MIS of 426 and related work by Ahn et al. (2020), claiming an average MIS of 418. Additionally to our own trained weights, we test the model weights provided in the official Intel-TreeSearch repository, but do not find any differences in the results.

Ahn et al. (2020) mention that they modified the official INTEL-TREESEARCH code. Unfortunately, neither the paper documents how exactly their queue pruning has been implemented, nor were the authors themselves able to provide us their modifications to the original INTEL-TREESEARCH repository when we contacted them. Hence, it remains unclear whether the difference in the SATLIB results between Ahn et al. (2020) and our experiments stems from how they implemented queue pruning, or from some unwanted side effects in their experiments (e.g., the reduction may have been still enabled). Furthermore, we unsuccessfully contacted Li et al. (2018) about our findings. As both the INTEL-TREESEARCH *and* our re-implementation DGL-TREESEARCH, which was written from scratch, exhibit this behavior, we suspect that there must be an undocumented modification or problem in the experiments that leads to Ahn et al. (2020) obtaining rather good results with the default configuration. Overall, neither the original code with the original weights, nor the original code with newly trained weights, nor our reimplementation are performing as originally claimed. In order to be as transparent as possible about our findings, we provide all of our code changes applied to the Intel repository as a patch and provide the entire source code of the re-implemented DGL-TREESEARCH as well.

Next, we analyze the replacement of the outputs of the GCN with random values. We start with the randomized default configuration, i.e., no techniques like reduction are enabled. In this case, we see that for all small random graphs, the tree search is still able to find solutions; however, the quality of these MIS seems to be slightly worse than the default results. For 80 % of the larger random graphs – except for ER graphs – the randomized tree search is not able to find solutions. As the default configuration is not able to find any solution for SATLIB, it is to be expected that the random configuration does not find any solution either.

For the other real-world graphs, the reddit datasets are the only datasets where the default configuration performed reasonably well. We find that the randomness here in fact increases the performance of the algorithm. Hence, the guidance by the GCN does not help our tree search algorithm; for some datasets, it is not able to generalize well (randomness beats the GCN), for others, there is no performance difference between random values and the GCN. If we focus on the configuration of the tree search that could be considered the *production* version[9], i.e., with at least reduction and local search enabled, we find that random outputs lead to identical performance, with very minor differences in the time until the final solution discovery. Overall, major contributing factors to solution quality are the reduction and local search by KAMIS, and in fact, the tree search algorithm is a "smarter brute-force" approach that does not gain any performance by being guided by a machine learning model. Instead, the tree-search is almost like a classic branch-and-bound solver, and techniques such as data reduction and weighted queue popping, that narrow the search space, lead to the discovery of good solutions.

Lastly, we briefly note that the experimental results are influenced by randomness, as the queue popping happens randomly; this can for example be seen in the two cases, where a solution was discovered in the default configuration of the DGL-TREESEARCH, but no solutions for the configuration with queue pruning enabled.

3.2 COMPARING TREE SEARCH SOLVERS TO OTHER SOLVERS

Having understood the implications of the various possible configurations of tree searches, next, we compare INTEL-TREESEARCH and DGL-TREESEARCH to KAMIS, a heuristic solver optimized for the MAXIMUM INDEPENDENT SET problem, the mathematical optimization tool GUROBI, and the reinforcement learning tool LEARNING WHAT TO DEFER. For LWD, we train for 20 000 iterations of proximal policy optimization on the SATLIB dataset, using the hyperparameters given for SATLIB in Table 5 (Appendix A.1) of Ahn et al. (2020).

The results are given in Table 2. As we can see, the sophisticated state-of-the-art solver KAMIS can solve almost all instances perfectly; for example, for the large ER graphs, the multithreaded INTEL-TREESEARCH has a slightly higher average MIS, and a faster time until a solution is found. For other graphs, especially the SATLIB dataset, KAMIS is very fast, while obtaining very high-quality results. Regarding the general-purpose solver GUROBI, we find that it performs similar to KAMIS on simple instances, however, on harder datasets such as SATLIB or VC-BM, we see that Gurobi

[9]Recall that the default version is not able to solve the SATLIB dataset without the help of reduction and local search techniques.

Table 2: Run times results of the modern MIS solver KAMIS, the optimization tool GUROBI, the reinforcement-learning based LEARNING WHAT TO DEFER, and the tree search algorithms in their default and full configuration. For the explanation of the various configuration flags, we refer to the caption of Table 1. All multithreaded experiments in this table use 8 threads, and all threads share a single GPU. Time limits are set equally to Table 1. We note that KAMIS and LWD always run single-threadedly, while GUROBI employs up to 8 threads, as needed. For LWD, we test the default configuration (`d`) and replace the output of the GNN with a random tensor (`rand`); the random graphs are executed with 32 iterations per episode, and all other graphs with 128 iterations per episode (c.f. Ahn et al. (2020)). We mark the configuration/solver that has the best average MIS in bold, and grey out configurations for which solutions were found for less than 20 % of all graphs.

		Intel		DGL		LwD		KaMIS	Gurobi
Graph	Nodes	d	r+ls+mt	d	r+qp+wp+ls+mt	d	+rand	default	default
ER	50-100	20.58 (0.98) 1.70 (500)	**20.83 (1.00)** 5.39 (500)	19.88 (0.95) 3.53 (500)	**20.83 (0.99)** 0.41 (500)	20.36 (0.97) 0.24 (500)	8.64 (0.41) 0.06 (500)	**20.83 (1.00)** 1.60 (500)	**20.83 (1.00)** 0.10 (500)
	700-800	39.90 (-) 12.00 (100)	**44.98 (-)** 11.90 (100)	37.13 (-) 13.63 (100)	44.32 (-) 22.97 (96)	33.65 (-) 0.37 (100)	8.85 (-) 0.12 (100)	44.57 (-) 31.28 (100)	37.79 (-) 30.01 (100)
BA	50-100	42.42 (0.99) 0.54 (500)	**42.43 (1.00)** 4.97 (500)	42.05 (0.99) 2.44 (500)	**42.43 (1.00)** 0.11 (500)	**42.43 (1.00)** 0.26 (500)	22.38 (0.53) 0.06 (500)	**42.43 (1.00)** 0.05 (500)	**42.43 (1.00)** 0.00 (500)
	700-800	420.89 (0.97) 3.09 (18)	**432.58 (1.00)** 5.31 (100)	389.65 (0.90) 16.50 (80)	**432.58 (1.00)** 0.69 (100)	432.57 (0.99) 0.28 (100)	187.79 (0.43) 0.10 (100)	**432.58 (1.00)** 0.06 (100)	**432.58 (1.00)** 0.01 (100)
HK	50-100	42.32 (0.99) 0.53 (500)	**42.33 (1.00)** 5.15 (500)	41.87 (0.98) 2.59 (500)	**42.33 (1.00)** 0.07 (500)	**42.33 (1.00)** 0.25 (500)	22.40 (0.53) 0.06 (500)	**42.33 (1.00)** 0.06 (500)	**42.33 (1.00)** 0.00 (500)
	700-800	417.89 (0.98) 4.23 (19)	**429.81 (1.00)** 5.26 (100)	390.64 (0.90) 19.66 (83)	**429.81 (1.00)** 0.42 (100)	429.77 (0.99) 0.27 (100)	188.94 (0.43) 0.09 (100)	**429.81 (1.00)** 0.08 (100)	**429.81 (1.00)** 0.01 (100)
WS	50-100	38.69 (0.99) 0.53 (500)	**38.70 (1.00)** 5.24 (500)	37.91 (0.97) 3.80 (500)	**38.70 (1.00)** 0.07 (500)	**38.70 (0.99)** 0.24 (500)	24.59 (0.63) 0.07 (500)	**38.70 (1.00)** 0.04 (500)	**38.70 (1.00)** 0.00 (500)
	700-800	- (-) -	**386.90 (1.00)** 5.49 (100)	- (-) -	**386.90 (1.00)** 0.39 (100)	**386.90 (1.00)** 0.28 (100)	220.07 (0.56) 0.09 (100)	**386.90 (1.00)** 0.05 (100)	**386.90 (1.00)** 0.00 (100)
HRG	50-100	33.62 (0.99) 3.24 (495)	**33.72 (1.00)** 5.11 (500)	32.80 (0.97) 5.50 (500)	**33.72 (1.00)** 0.06 (500)	**33.72 (1.00)** 0.26 (500)	19.51 (0.58) 0.07 (500)	**33.72 (1.00)** 0.06 (500)	**33.72 (1.00)** 0.00 (500)
	700-800	- (-) -	**304.21 (1.00)** 5.51 (100)	221.80 (0.75) 17.81 (5)	**304.21 (1.00)** 0.44 (100)	304.07 (0.99) 0.30 (100)	140.20 (0.46) 0.11 (100)	**304.21 (1.00)** 0.13 (100)	**304.21 (1.00)** 0.01 (100)
SATLIB	1 209-1 347	- (-) -	426.51 (0.99) 7.99 (500)	340.50 (0.79) 18.48 (2)	426.55 (0.99) 13.12 (500)	423.48 (0.99) 0.91 (500)	144.85 (0.33) 0.09 (500)	**426.59 (0.99)** 4.51 (500)	426.57 (0.99) 3.12 (500)
VC-BM	450-1 534	39.85 (0.90) 149.80 (40)	**45.10 (1.00)** 51.64 (40)	37.20 (0.85) 139.55 (40)	44.65 (0.99) 145.24 (40)	36.02 (0.85) 1.95 (40)	9.78 (0.25) 0.15 (40)	44.95 (0.99) 86.58 (40)	42.73 (0.99) 266.71 (40)
as-Caida	8 020-26 475	- (-) -	19387.47 (0.99) 14.49 (122)	- (-) -	**19387.90 (1.00)** 15.63 (122)	19387.47 (0.99) 3.65 (122)	7267.39 (0.37) 6.22 (122)	19387.47 (0.99) 2.23 (122)	19387.47 (0.99) 0.15 (122)
PPI	591-3 480	- (-) -	1002.71 (0.99) 8.73 (24)	269.00 (0.97) 23.56 (1)	1002.50 (0.99) 3.70 (24)	1001.54 (0.99) 1.67 (24)	251.62 (0.25) 0.28 (24)	**1002.83 (1.00)** 5.86 (24)	**1002.83 (1.00)** 8.16 (24)
REDDIT-B	46-2 283	146.06 (0.99) 5.94 (200)	**287.54 (1.00)** 5.37 (500)	186.31 (0.87) 11.81 (406)	**287.54 (1.00)** 0.26 (500)	**287.54 (1.00)** 0.90 (500)	147.79 (0.52) 0.08 (500)	**287.54 (1.00)** 0.18 (500)	**287.54 (1.00)** 0.00 (500)
REDDIT-5K	55-3 648	194.27 (0.99) 7.97 (37)	586.31 (0.99) 5.35 (500)	252.74 (0.88) 15.36 (152)	**586.32 (1.00)** 0.48 (500)	586.31 (0.99) 0.93 (500)	287.78 (0.49) 0.10 (500)	586.31 (0.99) 0.22 (500)	586.31 (0.99) 0.00 (500)
REDDIT-12K	41-897	128.61 (0.99) 3.99 (349)	**170.20 (1.00)** 5.20 (500)	136.05 (0.92) 9.53 (467)	**170.20 (1.00)** 0.17 (500)	**170.20 (1.00)** 0.86 (500)	91.45 (0.55) 0.07 (500)	**170.20 (1.00)** 0.10 (500)	**170.20 (1.00)** 0.00 (500)
wiki-RfA	11 380	- (-) -	**8111.00 (1.00)** 13.08	- (-) -	8096.00 (0.99) 19.95	8107.00 (0.99) 4.79	2243.00 (0.27) 3.07	**8111.00 (1.00)** 0.82	**8111.00 (1.00)** 1.57
wiki-Vote	7 115	- (-) -	**4866.00 (1.00)** 8.56	- (-) -	4864.00 (0.99) 11.20	4864.00 (0.99) 3.94	1430.00 (0.29) 2.18	**4866.00 (1.00)** 0.73	**4866.00 (1.00)** 0.87
PubMed	19 717	- (-) -	**15912.00 (1.00)** 12.91	- (-) -	15888.00 (0.99) 23.34	**15912.00 (1.00)** 3.34	5474.00 (0.34) 2.02	**15912.00 (1.00)** 0.16	**15912.00 (1.00)** 0.19
Cora	2 708	- (-) -	**1451.00 (1.00)** 5.63	- (-) -	**1451.00 (1.00)** 13.63	**1451.00 (1.00)** 2.74	644.00 (0.44) 1.25	**1451.00 (1.00)** 0.06	**1451.00 (1.00)** 0.03
Citeseer	3 264	- (-) -	**1808.00 (1.00)** 5.51	- (-) -	**1808.00 (1.00)** 14.62	**1808.00 (1.00)** 3.38	928.00 (0.51) 1.74	**1808.00 (1.00)** 0.06	**1808.00 (1.00)** 0.04
bitcoin-alpha	3 783	- (-) -	**2718.00 (1.00)** 6.10	- (-) -	2716.00 (0.99) 11.41	**2718.00 (1.00)** 2.37	1011.00 (0.37) 1.85	**2718.00 (1.00)** 0.21	**2718.00 (1.00)** 0.05
bitcoin-otc	5 881	- (-) -	4346.00 (0.99) 6.86	- (-) -	**4352.00 (1.00)** 12.47	4346.00 (0.99) 2.71	1574.00 (0.36) 2.21	4346.00 (0.99) 0.06	4346.00 (0.99) 0.08
ego-facebook	4 039	- (-) -	**1046.00 (1.00)** 14.34	- (-) -	- (-) -	1034.00 (0.98) 5.50	291.00 (0.27) 1.84	**1046.00 (1.00)** 19.86	**1046.00 (1.00)** 30.05
roadnet-berlin	61 204	- (-) -	29843.00 (0.99) 144.22	- (-) -	**29888.00 (1.00)** 68.22	29739.00 (0.99) 6.66	14520.00 (0.49) 2.17	29843.00 (0.99) 1.95	29843.00 (0.99) 22.72

takes significantly more time to find good solutions, and the average MIS is a little smaller (e.g., for large ER graphs, KAMIS achieves 44.57, compared to 37.79 for GUROBI).

We analyze LWD first in its default configuration and then, similar to the tree search, replace the output of the graph neural network with a random tensor. First, LWD is very fast, even though it requires neural network inference. Quality-wise, except for the VC-BM dataset, it always finds near-optimal solutions. Note that we did not use the local search or reduction techniques for LWD, which could further improve solution quality. When using random output instead of the neural network, unlike the tree search algorithms, the solution quality degrades noticeably. These promising results show that LWD did learn the solution structure of the MIS problem and is able to generalize over different datasets.

For tree search approaches, we see that state-of-the-art algorithmic techniques are required to find good solutions. As the fully-configured versions of the tree searches employ these KAMIS-internal routines, one can argue that the purely algorithmic solvers are the better choice, because the quality difference is negligible, while KAMIS is faster than the tree searches. Purely algorithmic solvers do not come with the overhead of a machine learning environment (training as well as execution are more complicated), and the important algorithmic techniques need to be developed in any case. However, the good results for LWD indicate that future work could focus on deep reinforcement learning solutions as well.

3.3 LARGE-SCALE GRAPHS

However, the question is whether these observations also hold true for large-scale graphs. To this end, we evaluate the solvers on random graph instances from 500 000 to 5 000 000 vertices, and on huge real-world graphs. Detailed results can be found in Appendix A in Table 3.

Overall, we observe similar performance characteristics on large-scale graphs. Only KAMIS and GUROBI are able to find solutions for all scenarios. KAMIS performs best, both with respect to solution quality and time required to find these solutions; in many cases, it is more than one order of magnitude faster than other solvers. Compared to the INTEL-TREESEARCH, the DGL-TREESEARCH time outs more often due to VRAM limits; the INTEL-TREESEARCH benefits from lower-level implementation using numpy arrays and sparse adjacency matrices, while the DGL-TREESEARCH employs the higher level Deep Graph Library graph abstraction, consuming more memory. For LWD, we see good results for graphs up to 500 000 vertices, but cannot verify the scalability up to 2 000 000 vertices of the original paper[10].

3.4 WEIGHTED GRAPHS

Having analyzed the solvers on unweighted graphs, we briefly want to see how the solvers perform on weighted graphs. Intuitively, the weighted problem is harder, because each vertex can have a different value; hence specialized reduction techniques have to be developed. Lamm et al. (2019) were the first to present such rules also for the weighted case that are implemented in KAMIS, however, only integer weights are currently supported. The results and details on the weighted graph generation are given in Appendix A in Table 4; note that INTEL-TREESEARCH does not support the weighted case, and while Ahn et al. (2020) evaluate LWD for MWIS, they do not provide the code to do so.

We observe that on HK, WS, and HRG graphs, where KAMIS is able to utilize its reduction techniques, it is able to instantaneously solve the weighted problem as well. However, on ER graphs, the reduction times out, showing that the suitability of the algorithmic techniques is graph-dependent. In the weighted case on ER graphs, the DGL-TREESEARCH with queue pruning enabled finds the best result. GUROBI finds solutions of similar quality to KAMIS very quickly. Overall, we see that the vanilla tree search has the advantage of being problem-agnostic, i.e., an extension to the weighted case was easily possible, while for the algorithmic solver, new reductions are needed, that currently cannot deal with some graphs. However, on graphs where the reductions are successful, KAMIS finds solutions of the highest quality, showing the trade-off between general and specialized solvers.

[10]As Ahn et al. (2020) do not explicitly state the hyperparameter configuration for their large-scale experiments and random graph generation, these experiments might not be directly comparible

4 Related Work

In this section, we briefly discuss some related work.

Solvers for Maximum Independent Set. Li et al. (2018) propose the Intel-TreeSearch, a supervised framework that determines an MIS over several steps while being guided in its choice of vertices by a GCN. As an alternative unsupervised approach, Khalil et al. (2017) propose S2V-DQN, in which they use Q-learning to solve the minimum vertex cover problem[11]. Building on their work, Ahn et al. (2020) propose Learning what to Defer (LwD), a reinforcement learning approach with a focus on scalability on large graphs. Similar to the tree search, LwD can mark multiple vertices as part of the MIS in a single step, which makes it faster compared to S2V-DQN, that only labels a single vertex in each step. A solver that does not utilize machine learning is KaMIS (Lamm et al., 2017; 2019; Hespe et al., 2019); we mention the recent publication by Hespe et al. (2021) introducing some new reduction rules. Note that all of these solvers are heuristics and hence only solve MIS approximately; exact solvers have been proposed by Jain & Seshadhri (2020); Xiao & Nagamochi (2017); Tomita et al. (2010), for example, and theory has been working on understanding why and in what models MIS poses to be difficult (Censor-Hillel et al., 2017).

Solvers for Other NP-hard Problems. Of course, other NP-hard problems that, unlike minimum vertex cover, are not closely related to maximum independent set, have also been subject to intensive research. The evolutionary computation community, for example, has researched various NP-hard routing problems, like the Vehicle Routing Problem (Berger & Barkaoui, 2003; Potvin, 2009) and the Multiple Routes problem (Böther et al., 2021). Another routing problem, the Travelling Salesman Problem (TSP), has already been tackled using neural networks in the 1980s (Hopfield & Tank, 1985). There have been two notable approaches by Bello et al. (2017) and Kool et al. (2019) that propose neural network-based solutions that are tailored towards the TSP. This means, that unlike the tree search approach, their solutions focus more on the problem at hand and cannot easily be generalized to other NP-hard problems. Last, we note the recent work by Nair et al. (2021) which aims at solving Mixed Integer Programs using Deep Learning.

5 Conclusion and Future Work

We present our comprehensive, open-source benchmark environment for the Maximum (Weighted) Independent Set problem. Using this environment, we run several experiments on both real-world and synthetic graphs of different sizes. Our analysis shows that guided tree searches, such as the Intel-TreeSearch by Li et al. (2018), owe their good results not to the trained neural network, but instead to the various techniques used to make a "better" brute-force algorithm. To verify this, we show that the GCN that guides the search can be replaced by random values without a noticeable performance impact. Furthermore, we are not able to reproduce the results of previous work in the default configuration of the algorithm and claim that without algorithmic techniques, the tree search algorithms are not able to solve hard MIS instances. We believe this to be an important insight for the community researching at the intersection of combinatorial optimization and machine learning. The benchmark suite lays the ground fur future reproducible evaluations for new MIS solvers, and the promising results for Learning What To Defer indicate that reinforcement learning is superior to supervised approaches. This might be kept in mind when developing solving techniques using machine learning in the future.

Ethics Statement

We adhere to the ICLR Code of Ethics and declare that we have no conflicts of interest. We tried to contact the authors of the related work, as stated in the respective sections.

Reproducibility Statement

As our results contradict previous work, we try to be as transparent as possible about our process. The solvers we use are documented in Section 2. The changes we apply to LwD and Intel-

[11]MVC is very related to the maximum independent set, as one can just flip the assignment.

TREESEARCH are supplied as a git patch with our benchmark suite that can be accessed via the anonymous repository[12] or in the supplementary material in OpenReview. The datasets we use are listed in Appendix D; we supply the code required to preprocess the datasets.

REFERENCES

Sungsoo Ahn, Younggyo Seo, and Jinwoo Shin. Learning what to defer for maximum independent sets. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, 2020.

Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1), 2002. doi: 10.1103/revmodphys.74.47.

Anaconda Inc. Anaconda software distribution, 2020. URL `https://docs.anaconda.com/`.

Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604), 2016. doi: 10.1038/533452a.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Jean Berger and Mohamed Barkaoui. A hybrid genetic algorithm for the capacitated vehicle routing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2003*, 2003. doi: 10.1007/3-540-45105-6_80.

Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Anton Krohmer, and Jonathan Striebel. Towards a systematic evaluation of generative network models. In *Proceedings of the International Workshop on Algorithms and Models for the Web Graph (WAW)*, 2018. doi: 10.1007/978-3-319-92871-5_8.

Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA)*, 2019. doi: 10.4230/LIPICS.ESA.2019.21.

Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65, 2017. doi: 10.1016/j.compenvurbsys.2017.05.004.

Maximilian Böther, Leon Schiller, Philipp Fischbeck, Louise Molitor, Martin S. Krejca, and Tobias Friedrich. Evolutionary minimization of traffic congestion. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2021. doi: 10.1145/3449639.3459307.

Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 2021. doi: 10.24963/ijcai.2021/595.

Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the congest model. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, 2017. doi: 10.4230/LIPICS.DISC.2017.10.

Qifeng Chen and Vladlen Koltun. Photographic image synthesis with cascaded refinement networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017. doi: 10.1109/iccv.2017.168.

Zihao Ding, Aniketh Reddy, and Aparna Joshi. Reproduciblity, 2020. URL `https://blog.ml.cmu.edu/2020/08/31/5-reproducibility/`.

[12] `https://anonymous.4open.science/r/mwis-benchmark`

Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5(1), 1960.

Majid Eskandarpour, Pierre Dejax, Joe Miemczyk, and Olivier Péton. Sustainable supply chain network design: An optimization-oriented review. *Omega*, 54, 2015. doi: 10.1016/j.omega.2015.01.006.

Tobias Friedrich. From graph theory to network science: The natural emergence of hyperbolicity (tutorial). In *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science (STACS)*, 2019. doi: 10.4230/LIPICS.STACS.2019.5.

Michael R. Garey and David S. Johnson. Strong NP-completeness results. *Journal of the ACM*, 25 (3), 1978. doi: 10.1145/322077.322090.

C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the 3rd Conference on Digital Libraries (DL)*, 1998. doi: 10.1145/276675.276685.

Gurobi Optimization LLC. Gurobi website, 2021. URL `https://www.gurobi.com/`.

Abner Guzmán-Rivera, Dhruv Batra, and Pushmeet Kohli. Multiple choice learning: Learning to produce multiple structured outputs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 25, 2012.

Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab. (LANL), Los Alamos, NM (United States), 2008.

William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31, 2017.

Demian Hespe, Christian Schulz, and Darren Strash. Scalable kernelization for maximum independent sets. *ACM Journal of Experimental Algorithmics*, 24(1), 2019. doi: 10.1145/3355502.

Demian Hespe, Sebastian Lamm, and Christian Schorr. Targeted Branching for the Maximum Independent Set Problem. In *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA)*, 2021. doi: 10.4230/LIPIcs.SEA.2021.17.

Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2), 2002. doi: 10.1103/physreve.65.026107.

Holger Hoos and Thomas Stützle. Satlib: An online resource for research on sat. In *SAT 2000*. IOS Press, 2000.

John J. Hopfield and David W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52(3), 1985. doi: 10.1007/BF00339943.

John P. A. Ioannidis. Why most published research findings are false. *PLoS Medicine*, 2(8), 2005. doi: 10.1371/journal.pmed.0020124.

Shweta Jain and C. Seshadhri. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining (WSDM)*, 2020. doi: 10.1145/3336191.3371839.

Sayash Kapoor and Arvind Narayanan. Irreproducibility in machine learning, 2020. URL `https://reproducible.cs.princeton.edu/`.

Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Springer US, 1972. doi: 10.1007/978-1-4684-2001-2_9.

Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3), 2010. doi: 10.1103/physreve.82.036106.

Srijan Kumar, Francesca Spezzano, V. S. Subrahmanian, and Christos Faloutsos. Edge weight prediction in weighted signed networks. In *Proceedings of the 16th IEEE International Conference on Data Mining (ICDM)*, 2016. doi: 10.1109/icdm.2016.0033.

Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and V.S. Subrahmanian. REV2: fraudulent user prediction in rating platforms. In *Proceedings of the 11th International Conference on Web Search and Data Mining (WSDM)*, 2018. doi: 10.1145/3159652.3159729.

Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Finding near-optimal independent sets at scale. *Journal of Heuristics*, 23(4), 2017. doi: 10.1007/s10732-017-9337-x.

Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *Proceedings of the 21st Symposium on Algorithm Engineering and Experiments (ALENEX)*, 2019. doi: 10.1137/1.9781611975499.12.

Jure Leskovec and Julian Mcauley. Learning to discover social circles in ego networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 25, 2012.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceeding of the 11th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005. doi: 10.1145/1081870.1081893.

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1), 2009. doi: 10.1080/15427951.2009.10129177.

Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI)*, 2010. doi: 10.1145/1753326.1753532.

Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31, 2018.

Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2), 2000. doi: 10.1023/a:1009953814988.

Raymond E. Miller and David E. Muller. A problem of maximum consistent subsets. *IBM Research Report RC-240*, 1960.

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks. *arXiv preprint 2012.13349*, 2021.

OpenStreetMap Contributors. Planet dump retrieved from https://planet.osm.org. `https://www.openstreetmap.org`, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.

Jannik Peters, Daniel Stephan, Isabel Amon, Hans Gawendowicz, Julius Lischeid, Lennart Salabarria, Jonas Umland, Felix Werner, Martin S. Krejca, Ralf Rothenberger, Timo Kötzing, and Tobias Friedrich. Mixed integer programming versus evolutionary computation for optimizing a hard real-world staff assignment problem. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS)*, 2019.

Jean-Yves Potvin. State-of-the art review—evolutionary algorithms for vehicle routing. *INFORMS Journal on Computing*, 21(4), 2009. doi: 10.1287/ijoc.1080.0312.

Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL `https://networkrepository.com`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint 1707.06347*, 2017.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3), 2008. doi: 10.1609/aimag.v29i3.2157.

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 *Proceedings of the International Workshop on Algorithms and Computation (WALCOM)*, 2010. doi: 10.1007/978-3-642-11440-3_18.

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

Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684), 1998. doi: 10.1038/30918.

Robert West, Hristo S. Paskov, Jure Leskovec, and Christopher Potts. Exploiting social network structure for person-to-person sentiment analysis. *Transactions of the Association for Computational Linguistics*, 2, 2014. doi: 10.1162/tacl_a_00184.

Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255, 2017. doi: 10.1016/j.ic.2017.06.001.

Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9), 2007. doi: 10.1016/j.artint.2007.04.001.

Pinar Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2015. doi: 10.1145/2783258.2783417.

Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1), 2013. doi: 10.1007/s10115-013-0693-z.

A Additional Tables

In this section, you can find the tables omitted in the main paper due to space constraints.

Table 3: Results of the modern MIS solver KaMIS, the optimization tool Gurobi, the reinforcement-learning based Learning What To Defer, and the tree search algorithms in their full configuration, on huge random graphs and huge real world networks. We do not test huge ER graphs due to the time complexity of generating them, and do not test BA graphs as they are similar to HK graphs. For the explanation of the various configuration flags, we refer to the caption of Table 1. All multithreaded experiments in this table use 8 threads, and all threads share a single GPU. The time limit for all graphs was set to 3 hours. For LwD, the algorithm is executed with with 128 iterations per episode (c.f. Ahn et al. (2020)). We mark the configuration that has the best average MIS in bold. Configurations that consumed too much VRAM or DRAM are noted as out of memory (OOM).

		Intel	DGL	LwD	KaMIS	Gurobi
Graph	Nodes	r+ls+mt	r+qp+wp+ls+mt	default	default	default
HK	500 000	**289486.00 (-)** 242.91	**289486.00 (-)** 177.12	**289486.00 (-)** 145.40	**289486.00 (-)** 0.27	**289486.00 (-)** 26.86
	2 000 000	**1157643.00 (-)** 1064.53	OOM	OOM	**1157643.00 (-)** 1.15	**1157643.00 (-)** 154.14
	5 000 000	**2892242.00 (-)** 1207.98	OOM	OOM	**2892242.00 (-)** 3.80	**2892242.00 (-)** 446.81
WS	500 000	**258887.00 (-)** 219.57	**258887.00 (-)** 171.97	**258887.00 (-)** 61.28	**258887.00 (-)** 0.22	**258887.00 (-)** 5.41
	2 000 000	**1035715.00 (-)** 923.82	**1035715.00 (-)** 740.14	OOM	**1035715.00 (-)** 0.95	**1035715.00 (-)** 28.48
	5 000 000	**2589519.00 (-)** 1128.76	OOM	OOM	**2589519.00 (-)** 2.82	**2589519.00 (-)** 76.26
HRG	500 000	**210950.00 (-)** 374.53	**210950.00 (-)** 216.09	210930.00 (-) 834.99	**210950.00 (-)** 0.78	210949.00 (-) 47.20
	2 000 000	**843618.00 (-)** 1114.92	OOM	OOM	**843618.00 (-)** 3.51	843616.00 (-) 374.15
	5 000 000	**2110401.00 (-)** 1866.42	OOM	OOM	**2110401.00 (-)** 8.62	2110400.00 (-) 897.20
DBLP	317 080	**152131.00 (1.00)** 142.15 (2)	**152131.00 (1.00)** 112.32 (2)	**152131.00 (1.00)** 236.98 (2)	**152131.00 (1.00)** 0.30 (2)	**152131.00 (1.00)** 7.15 (2)
roadnet-pennsylvania	1 088 092	OOM	OOM	OOM	533625.00 (-) 17.49 (2)	**533628.00 (-)** 3662.87 (2)
ego-gplus	107 614	**57394.00 (-)** 399.20 (2)	OOM	56794.00 (-) 275.58 (2)	**57394.00 (-)** 115.76 (2)	57321.00 (-) 10802.52 (2)
web-google	875 713	**529138.00 (1.00)** 663.18 (2)	OOM	528725.00 (0.99) 822.14 (2)	**529138.00 (1.00)** 10.67 (2)	**529138.00 (1.00)** 83.30 (2)

Table 4: Results of the MIS solver KaMIS, the optimization tool Gurobi, and the DGL-TreeSearch in various configurations on weighted random graphs. The weights have been sampled per vertex from a normal distribution $\mathcal{N}$ with $\mu = 100, \sigma = 30$, capped to be larger than 0, and rounded to integer values. For the explanation of the various configuration flags, we refer to the caption of Table 1. All multithreaded experiments in this table use 8 threads, and all threads share a single GPU. The time limit for all graphs was set to 30 seconds. We mark the configuration that has the best average MIS in bold, and grey out configurations for which solutions were found for less than 20 % of all graphs. Similar to the unweighted experiments, we run the solvers on 500 graphs per type. Configurations where there was a timeout due to KaMIS' reduction techniques are marked as TO.

		DGL								KaMIS	Gurobi
Graph	Nodes	d	+r	+ls	qp	+wp	+r	+ls	+mt	default	default
ER	700-800	3765.08 (-) 18.80 (500)	TO	TO	**3792.71 (-)** 18.01 (500)	3571.05 (-) 11.65 (500)	TO	TO	TO	TO	3709.68 (-) 30.01 (500)
HK	700-800	39469.48 (0.87) 20.11 (364)	42026.61 (0.93) 1.02 (500)	45034.05 (0.99) 0.68 (500)	39128.12 (0.87) 20.18 (329)	40441.76 (0.89) 11.89 (500)	42026.61 (0.93) 0.97 (500)	45034.01 (0.99) 0.75 (500)	45033.46 (0.99) 0.92 (500)	**45036.91 (0.99)** 0.00 (500)	**45036.91 (0.99)** 0.01 (500)
WS	700-800	- (-) -	36010.38 (0.87) 3.13 (500)	41160.71 (0.99) 5.38 (500)	37958.50 (0.92) 14.77 (2)	36904.74 (0.89) 15.17 (439)	36010.51 (0.87) 3.41 (500)	41159.49 (0.99) 6.13 (500)	41147.84 (0.99) 10.19 (500)	**41196.43 (1.00)** 0.00 (500)	**41196.43 (1.00)** 0.01 (500)
HRG	700-800	23641.88 (0.70) 23.25 (24)	33366.88 (0.99) 0.62 (500)	33696.93 (0.99) 0.57 (500)	22583.27 (0.68) 24.89 (22)	24497.41 (0.72) 17.64 (482)	33366.88 (0.99) 0.60 (500)	33696.93 (0.99) 0.54 (500)	33696.93 (0.99) 0.57 (500)	**33696.96 (1.00)** 0.00 (500)	**33696.96 (1.00)** 0.01 (500)

B TREE SEARCH

In the following, we describe the guided tree search approach introduced by Li et al. (2018) in more detail, and explain our additional modifications (e.g., the queue pruning). A pseudocode description of the algorithm is found in Algorithm 1; for a visualization, we refer to Figure 1 of Li et al. (2018). The core element of the tree search is a queue[13] P of partial solutions $S \in \{0, 1, ?\}^{|V|}$ to the MIS problem, i.e., labelings of the graph at hand marking each vertex as either *included* in the MIS (1), *excluded* (0), or *unlabeled* (a decision about that vertex is still to be made, ?). To be a valid (partial) solution to the MIS problem, each element in the queue fulfills the constraint that no two adjacent vertices can both be included. Furthermore, all vertices adjacent to an *included* one must be *excluded*.

Given a graph $G = (V, E)$, we start with an empty solution, i.e., f.a. $v \in V$, $S_v = ?$. In each step of the tree search, a partial labeling is popped from the queue (random_pop), and the *residual graph* G_{residual} consisting only of the unlabeled vertices is constructed (keeping only those edges between unlabeled vertices). Next, we obtain a predefined number m of *probability maps*, each of which contains a value for all vertices of the residual graph, posing the "probability of being in the MIS". This is done using a function $\texttt{gen\_probmaps} : \mathfrak{G} \to [0, 1]^{|V| \times m}$, that can take arbitrary unweighted graphs $G \in \mathfrak{G}$ as input. In the default case, this function calls the trained GCN, which outputs its assignments from vertices to probabilities, i.e., $\mathsf{GCN}(G) \in [0, 1]^{|V| \times m}$.

From each of these probability maps, a new partial solution is derived as follows: The probabilities of the maps are sorted in descending order. The vertex with the highest probability gets labeled as *included*, and all adjacent vertices as *excluded*. This step is repeated until we would have to label an already *excluded* vertex as *included*, in which case we break, and add the partial solution to the queue. In case all vertices are labeled we obtained a full solution, which we yield and do not add back into the queue. This procedure is repeated for all probability maps. A non-pseudocode implementation – like DGL-TREESEARCH – could, for example, choose the largest independent set that is yielded to obtain a potential MIS.

Furthermore, there are several modifications (flags) discussed in Section 3.1. In the following, we briefly discuss how these flags impact the tree search algorithm.

- **Reduction.** Reduction or graph kernelization is an addition that uses algorithmic techniques to find vertices which (provably) *have to* be in the independent set. After determining the residual graph, before generating the probability maps, we optionally can determine whether some vertices can be labeled due to the reduction rules, and then continue to determine the probability maps with an even smaller residual graph. Lamm et al. (2017) and Hespe et al. (2019) provide reduction rules for the unweighted case, which both INTEL-TREESEARCH and DGL-TREESEARCH access using a Python-C++ interface. Furthermore, for the MAXIMUM WEIGHTED INDEPENDENT SET problem, DGL-TREESEARCH uses the recently proposed reduction rules by Lamm et al. (2019).
- **Local Search.** A local search is an addition based on small mutations which try to further improve a full solution before it is yielded. It can be understood as fine-tuning a solution. The INTEL-TREESEARCH and DGL-TREESEARCH use the local search implementation by Lamm et al. (2017) and Hespe et al. (2019), and for the weighted case, DGL-TREESEARCH uses the local search by Lamm et al. (2019).
- **Multithreading.** To find the MIS faster, one can use multiple threads that search for independent sets. The parallelization idea is very simple, each thread just follows the same procedure as stated in Algorithm 1. In the INTEL-TREESEARCH, the threads share the queue P, making push and pop operations a critical section. Because we want to avoid this critical section, and due to issues with shared state in Python multiprocessing, in the DGL-TREESEARCH, we give each thread its own queue, and first initialize enough partial solutions, such that each thread has one partial solution to start on.
- **Queue Pruning.** As we see in Section 3.1, on difficult datasets like SATLIB, the tree search almost cannot find any solution. Furthermore, in their experiments, Ahn et al. (2020) state that the INTEL-TREESEARCH runs out of memory, something that we did not observe

[13] We use the term *queue* to stay consistent with other works. However, P is not a traditional LIFO queue, but a list (infinite-sized array).

Algorithm 1: Guided Treesearch Algorithm to find MIS (adjusted from Li et al. (2018)). For a solution S and vertex $u \in V$, we denote by S_u the current status of u (included, excluded, unlabeled). For a more detailed description, we refer to Appendix B.

Input: Graph $G = (V, E)$, Function $\texttt{gen\_probmaps} : \mathfrak{G} \to [0, 1]^{|V| \times m}$.
Output: Independent sets $S_i \subseteq V$.

$P \leftarrow \{(?, ?, \dots, ?) \in \{0, 1, ?\}^{|V|}\}$;
while *P is not empty* **do**
  $S \leftarrow \mathsf{random\_pop}(P)$;
  $V_{\mathsf{residual}} \leftarrow \{u \in V \mid S_u = \,?\}$;
  $G_{\mathsf{residual}} \leftarrow (V_{\mathsf{residual}}, \{(u, v) \in E \mid u, v \in V_{\mathsf{residual}}\})$;
  for $M \in \mathsf{gen\_probmaps}(G_{\mathsf{residual}})$ **do**
    $S' \leftarrow S$;
    for $u \in V_{\mathsf{residual}}$ *sorted by descending probability in M* **do**
      if $S'_u = 0$ **then**
        break;
      $S'_u \leftarrow 1$;
      for $v \in V_{\mathsf{residual}}$ *adjacent to u* **do**
        $S'_v \leftarrow 0$;
    if $\forall u \in V : S'_u \neq \,?$ **then**
      yield S';
    else
      append S' to P;

for medium-sized graphs. Queue pruning is a technique used to tackle these two issues. Unfortunately, we could not obtain the information on how the queue pruning approach by Ahn et al. (2020) is implemented via private communication, hence we propose our own solution. We set a maximum number of elements on our queue P, and after appending a new partial solution to the queue, we remove the oldest elements (at position 0) from the queue, until the queue is small enough again. The intuition is that we find more solutions faster, because there is a higher chance to pop an almost labeled (more recent) solution from the queue, and we also limit the memory used. Queue pruning was not proposed in the original paper by Li et al. (2018).

- **Weighted Queue Pop.** By default, the $\mathsf{random\_pop}$ used in Algorithm 1 chooses an element from the queue uniformly at random. Weighted Queue Pop, on the other hand, shifts the probability of each element to be chosen according to how many unlabeled vertices are left in it, favoring fewer unlabeled vertices. Similar to queue pruning, the intuition is to find fully labeled solutions faster while keeping some randomness in the popping behavior. It can be freely combined with queue pruning or used on its own. To the best of our knowledge, no other paper has proposed this addition to the tree search algorithm yet.
- **Random Values as Probability Maps.** This flag changes $\mathsf{gen\_probmaps}$ to a random generator, effectively replacing the GCN output with probability values chosen uniformly at random.

We note that the original implementation by Li et al. (2018) does not exactly follow our description and Algorithm 1. We note differences between their paper and the published code in the next section.

C Remarks on the Intel-TreeSearch Implementation

In this section, we note some important facts to keep in mind when dealing with the original tree search implementation, next to the rather difficult understandability and maintainability of the code. First, for difficult instances, with reduction and local search both disabled, the search queue appears to grow indefinitely without ever finding a single solution. To circumvent this effect and to be able to find solutions, it might be necessary to limit the search queue to a fixed maximum size, as observed by Ahn et al. (2020). They furthermore stress the importance of pruning to limit memory usage. We discuss our approach to queue pruning in Appendix B.

Second, in the paper, it is stated that the GCN is trained for 200 epochs. However, in the implementation, one epoch iterates over just 2000 randomly chosen samples of the 38000 samples in the training set[14]. This is not what you would expect given the term "epoch". Furthermore, the original code has some hardcoded values (e.g., sometimes it expects the number of input graphs to be flexible, and sometimes it is hardcoded within the same file). Our patch provided in the benchmark suite fixes these issues.

Third, the multi-threaded variant of the algorithm (`demo_parallel.py`) uses just one randomly chosen probability map of the GCN outputs instead of all. This makes this variant's performance in terms of search steps per second, appear to scale superlinearly, and furthermore is not documented in the paper[15], and needs to be kept in mind for evaluation.

D DATASETS

In this section, we give some details on the datasets, groups of datasets, and random graphs models used in this paper. For all datasets, we treat all graphs as undirected. The data generation module of our benchmarking suite integrates the download/generation and conversion of these graphs. We delete all self-loops, if there are any, because the solvers deal with self-loops differently (for example, with silent failures, errors, or just ignoring the vertex).

Erdős-Rényi. This well-known random graph model by Erdős & Rényi (1960) connects each pair of vertices with probability p. We use $p = 0.15$ and for testing, generate 500 graphs with 50 to 100 vertices and 100 graphs with 700 to 800 vertices.

Barabási-Albert. This random graph model by Albert & Barabási (2002) iteratively adds nodes, connecting them to m already existing nodes. We use $m = 2$ and for testing, generate 500 graphs with 50 to 100 vertices and 100 graphs with 700 to 800 vertices.

Holme-Kim. A random graph model by Holme & Kim (2002) similar to the BA-model, with an extra step for each randomly created edge that creates a triangle with probability p. We use $m = 2$ and $p = 0.05$ and for testing, generate 500 graphs with 50 to 100 vertices and 100 graphs with 700 to 800 vertices. For the large-scale experiments, we test on one graph per fixed amount of vertices, c.f. Table 3.

Watts-Strogatz. A random graph model by Watts & Strogatz (1998) that starts with a well-structured ring-lattice with mean degree k and in the following step replaces each edge with probability p with another edge sampled uniformly at random. This way it tries to keep "small-world properties" while maintaining a random structure similar to Erdős-Rényi graphs. We use $k = 2$ and $p = 0.15$. For the large-scale experiments, we test on one graph per fixed amount of vertices, c.f. Table 3.

Hyperbolic Random Graph. A random graph model by Krioukov et al. (2010), which generates graphs by randomly putting nodes in a disk in the hyperbolic plane, and connecting them if their (hyperbolic) distance is below a certain threshold. Recent works have it described to be the best model for modeling real-world networks, due to their heterogeneity and interdependency (Friedrich, 2019). As generation parameters, we use $\alpha = 0.75$, $T = 0.1$, $deg = 10$ and for testing, generate 500 graphs with 50 to 100 vertices and 100 graphs with 700 to 800 vertices. For the large-scale experiments, we test on one graph per fixed amount of vertices, c.f. Table 3.

SATLIB. This is a well-known set of SAT instances in CNF commonly used as a benchmark for SAT solvers (Hoos & Stützle, 2000). A SAT instance can easily be reduced to a graph, which has a MIS as large as the number of clauses, should the SAT instance be satisfiable. In short, for each

[14] `https://github.com/isl-org/NPHard/blob/5fc770ce1b1daee3cc9b318046f2361611894c27/train.py#L92`

[15] Algorithm 3 in Appendix C.1 of the paper contains an unbound variable m responsible for this ambiguity.

clause a clique is created, each node of which stands for a variable or its negation, e.g., x or $\neg x$. Afterwards, for each variable x, all its nodes are connected to nodes referring to its negation $\neg x$. We use the "Random-3-SAT Instances with Controlled Backbone Size" dataset[16], which consists of 40 000 instances, of which we train on 39 500 and test on 500. Each instance has between 403 to 449 clauses.

PPI. The PPI dataset, introduced by Hamilton et al. (2017), contains of 24 graphs whose nodes describe proteins and edges describe interactions between them. They contain 591 to 3 480 nodes. We use the data set provided by DGL[17], as it provides a more easily understandable separation between the graphs than the original source by GraphSAGE[18].

REDDIT. The REDDIT datasets, introduced by Yanardag & Vishwanathan (2015), contain graphs constructed from online discussion threads on Reddit[19]. Vertices represent users and edges represent at least one response by one of the users to the other. The difference between REDDIT-BINARY, REDDIT-MULTI-5K, and REDDIT-MULTI-12K is how many subreddits have been crawled, and whether the classification task that is usually associated with these datasets is binary or multi-class classification. We obtain the data from TU Dortmund's data set collection[20]. For each dataset, we sample 500 graphs for testing, and they contain 41-3 648 nodes.

as-Caida. The as-Caida dataset of autonomous systems, introduced by Leskovec et al. (2005), consists of 122 graphs derived from a set of RouteViews BGP table snapshots. They contain 8 020 to 26 475 nodes. We obtain the data from Stanford's SNAP repository[21].

Citation. The Citation dataset consists of the three citation network graphs *Cora* (McCallum et al., 2000), *Citeseer* (Giles et al., 1998) and *PubMed* (Sen et al., 2008). We obtain all data from NetworkRepository (Rossi & Ahmed, 2015).

DBLP-Coauthorship. This dataset contains the *com-DBLP*[22] (Yang & Leskovec, 2013) graph. Vertices represent authors and an edge exists if two authors have published at least one paper together.

Road Networks. The Road Networks dataset contains two road networks of different sizes. The network *roadNet-PA* obtained from SNAP[23] represents the state of Pennsylvania (Leskovec et al., 2009). The other network maps the city of Berlin, Germany, and is extracted from OpenStreetMap (OpenStreetMap Contributors, 2017) using OSMnx (Boeing, 2017).

Social Networks. The Social Networks dataset contains of 6 social network graphs we gathered. Vertices represent users and edges represent some kind of relationship between them. The exact kind of relationship depends on the graph. The *ego-Facebook*[24] and *ego-Gplus*[25] graphs represent "friendships" between users (Leskovec & Mcauley, 2012). The *bitcoin-otc*[26] and *bitcoin-alpha*[27] graphs represent Bitcoin's web-of-trust network (Kumar et al., 2016; 2018). The *wiki-Vote*[28] and *wiki-RfA*[29] graphs are derived from interactions in the governing process of Wikipedia (Leskovec et al., 2010; West et al., 2014).

[16] https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/CBS/descr_CBS.html
[17] https://docs.dgl.ai/en/0.6.x/_modules/dgl/data/ppi.html
[18] http://snap.stanford.edu/graphsage/ppi.zip
[19] https://reddit.com
[20] https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets
[21] https://snap.stanford.edu/data/as-caida.html
[22] https://snap.stanford.edu/data/com-DBLP.html
[23] https://snap.stanford.edu/data/roadNet-PA.html
[24] https://snap.stanford.edu/data/ego-Facebook.html
[25] https://snap.stanford.edu/data/ego-Gplus.html
[26] https://snap.stanford.edu/data/soc-sign-bitcoin-otc.html
[27] https://snap.stanford.edu/data/soc-sign-bitcoin-alpha.html
[28] https://snap.stanford.edu/data/wiki-Vote.html
[29] https://suitesparse-collection-website.herokuapp.com/SNAP/wiki-RfA

VC-Benchmark (VC-BM). A dataset consisting of 40 graphs of different sizes from 450 to 1 534 nodes, on which finding the maximum independent set is synthetically made hard (Xu et al., 2007). Li et al. (2018) call this dataset BUAA-MC. As the original download[30] is not available anymore, we use a GitHub mirror[31].

[30] http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm
[31] https://unsat.github.io/npbench/vertexcovering.html