Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21)
Survey Track

Combinatorial Optimization and Reasoning with Graph Neural Networks

Quentin Cappart[1], **Didier Chételat**[2], **Elias B. Khalil**[3], **Andrea Lodi**[2], **Christopher Morris**[2*] and **Petar Veličković**[4]

[1]Department of Computer Engineering and Software Engineering, Polytechnique Montréal,
[2]CERC in Data Science for Real-Time Decision-Making, Polytechnique Montréal,
[3]Department of Mechanical & Industrial Engineering, University of Toronto,
[4]DeepMind

Abstract

Combinatorial optimization is a well-established area in operations research and computer science. Until recently, its methods have mostly focused on solving problem instances in isolation, ignoring the fact that they often stem from related data distributions in practice. However, recent years have seen a surge of interest in using machine learning, especially graph neural networks, as a key building block for combinatorial tasks, either directly as solvers or by enhancing the former. This paper presents a conceptual review of recent key advancements in this emerging field, aiming at researchers in both optimization and machine learning.

1 Introduction

Combinatorial optimization (CO) has developed into an interdisciplinary field spanning optimization, operations research, discrete mathematics, and computer science, with many critical real-world applications such as vehicle routing or scheduling [Korte and Vygen, 2012]. Intuitively, CO deals with problems that involve optimizing a cost (or objective) function by selecting a subset from a finite set, with the latter encoding constraints on the solution space. Although CO problems are generally hard from a complexity theory standpoint due to their discrete, non-convex nature [Karp, 1972], many of them are routinely solved in practice. Historically, the optimization and theoretical computer science communities have been focusing on finding optimal [Korte and Vygen, 2012], heuristic [Boussaïd *et al.*, 2013], or approximate [Vazirani, 2010] solutions for individual problem instances. However, in many practical situations of interest, one often needs to solve problem instances which share certain characteristics or patterns. Hence, data-dependent algorithms or machine learning approaches, which may exploit these patterns, have recently gained traction in the CO field [Bengio *et al.*, 2021; Kotary *et al.*, 2021]. The promise here is that by exploiting common patterns in the given instances, one can develop faster algorithms for practical cases. Due to the discrete nature of most CO problems and the prevalence of network data in the real world, graphs (and their relational generalizations) are a central object of study in the CO field. In fact, from the 21 NP-complete problems identified by Karp [1972], ten are decision versions of graph optimization problems, e.g., the *travelling saleperson problem* (TSP). Most of the other ones, such as the set covering problem, can also be modeled over graphs. Moreover, the interaction between variables and constraints in constraint optimization problems naturally induces a bipartite graph, i.e., a variable and constraint share an edge if the variable appears with a non-zero coefficient in the constraint. These graphs commonly exhibit patterns in their structure and features, which machine learning approaches should exploit.[1]

What Are the Challenges for ML? There are several critical challenges in successfully applying machine learning methods within CO, especially for problems involving graphs. Graphs have no unique representation, i.e., renaming or reordering the nodes does not result in different graphs. Hence, for any machine learning method dealing with graphs, taking into account invariance to permutation is crucial. Combinatorial optimization problem instances, especially those arising from the real world, are large and usually sparse. Hence, the employed machine learning method must be scalable and sparsity aware. Simultaneously, the employed method has to be expressive enough to detect and exploit the relevant patterns in the given instance or data distribution while still generalizing beyond its training data. The machine learning method should be capable of handling auxiliary information, such as objective and user-defined constraints. Most of the current machine learning approaches are within the supervised regime. That is, they require a large amount of training data to optimize the parameters of the model. In the context of CO, this means solving many possibly hard problem instances, which might prohibit the application of these approaches in real-world scenarios.

How Do GNNs Solve These Challenges? Graph neural networks (GNNs) compute a vectorial representation, e.g., a real vector, of each node in the input graph by iteratively aggregating features of neighboring nodes. By parameterizing this aggregation step, the GNN is trained in an end-to-end fashion against a loss function. The promise here is that the learned vector representation encodes crucial graph structures that help solve a CO problem more efficiently. GNNs are invariant by design, i.e., they automatically exploit the invariances inherent to the given instance. Due to their local nature, by aggregating neighborhood information, GNNs naturally exploit sparsity, leading to more scalable models on sparse inputs. Moreover, although scalability

[*]Corresponding author

[1]See [Cappart *et al.*, 2021] for an extended version of the present work.

is still an issue, they scale linearly with the number of edges and employed parameters, while taking multi-dimensional node and edge features into account [Gilmer *et al.*, 2017], naturally exploiting cost and objective function information. However, the data-efficiency question is still largely open. Although GNNs have clear limitations [Morris *et al.*, 2019], they have already proven to be useful in the context of combinatorial optimization, either predicting a solution directly or as an integrated component of an existing solver. We will extensively investigate both of these aspects within our survey.

Going Beyond Classical Algorithms The previous discussion mainly dealt with the idea of machine learning approaches, especially GNNs, replacing and imitating classical combinatorial algorithms or parts of them, potentially adapting better to the specific data distribution of naturally-occurring problem instances. However, classical algorithms heavily depend on human-made pre-processing or feature engineering by abstracting raw, real-world inputs. In the long-term, machine learning approaches can further enhance the CO pipeline, from raw input processing to aid in solving abstracted CO problems in an end-to-end fashion. Several viable approaches in this direction have been proposed recently, and we will survey them in detail.

1.1 Preliminaries

Here, we introduce notation and give the necessary background on CO and GNNs.

Notation Let $[n] = \{1, \dots, n\} \subset \mathbb{N}$ for $n \geq 1$, and let $\{\!\!\{ \dots \}\!\!\}$ denote a multiset. For a (finite) set S, we denote its *power set* as 2^S. A *graph* G is a pair (V, E) with a *finite* set of *nodes* V and a set of *edges* $E \subseteq V^2$. We denote the set of nodes and the set of edges of G by $V(G)$ and $E(G)$, respectively. The *neighborhood* of v in $V(G)$ is denoted by $N(v) = \{u \in V(G) \mid (v, u) \in E(G)\}$.

Combinatorial Optimization Intuitively, CO aims at choosing a subset from a finite set, optimizing a *cost function*. Formally, an instance of a *CO problem* is a tuple (Ω, F, w), where Ω is a *finite* set, $F \subseteq 2^{\Omega}$ is the set of *feasible* solutions, $w\colon \Omega \to \mathbb{Q}$ assigns *costs*, inducing the cost function $c(S) = \sum_{\omega \in S} w(\omega)$ for S in F. Consequently, CO deals with selecting an element S^* (*optimum*) in F that minimizes $c(S^*)$ over F.

Graph Neural Networks Intuitively, GNNs compute a vectorial representation, i.e., a d-dimensional vector, for each node in a graph by aggregating information from neighboring nodes. Each layer of a GNN aggregates neighbors' features of each node v and then passes this aggregated information on to the next layer. Following [Gilmer *et al.*, 2017], in full generality, a new feature $f^{(t)}(v)$ is computed as

$$f^{W_1}_{\text{merge}}\Big(f^{(t-1)}(v), f^{W_2}_{\text{aggr}}\big(\{\!\!\{ f^{(t-1)}(w) \mid w \in N(v) \}\!\!\}\big)\Big),$$

where $f^{W_1}_{\text{aggr}}$ aggregates over the multiset of neighborhood features and $f^{W_2}_{\text{merge}}$ merges the node's representations from step $(t-1)$ with the computed neighborhood features. Both $f^{W_1}_{\text{aggr}}$ and $f^{W_2}_{\text{merge}}$ may be arbitrary differentiable functions with parameters W_1 and W_2.

2 GNNs for Combinatorial Optimization

Given that many practically relevant CO problems are NP-hard, it is helpful to characterize algorithms for solving them as prioritizing one of two goals. The *primal* goal of finding good feasible solutions, and the *dual* goal of certifying optimality or proving infeasibility. In both cases, GNNs can serve as a tool for representing problem instances, states of an iterative algorithm, or both. Coupled with an appropriate ML paradigm, GNNs have been shown to guide exact and heuristic algorithms towards finding good feasible solutions faster (Section 2.1) or to guide certifying optimality or infeasibility more efficiently (Section 2.2). Beyond using standard GNN models for CO, the emerging paradigm of algorithmic reasoning provides new perspectives on designing and training GNNs that satisfy natural invariants and properties (Section 2.3).

2.1 On the Primal Side: Finding Feasible Solutions

Here, we will discuss various approaches that leverage GNNs in the primal setting. First, we discuss approaches that find a solutions from scratch, followed by approaches replacing specific components in solvers.

Learning Heuristics from Scratch

The works in this section attempt to construct a feasible solution "from scratch", not using a combinatorial solver to help the machine learning model. In [Khalil *et al.*, 2017], GNNs were leveraged for the first time in the context of graph optimization problems. Here, the GNN served as the function approximator for the value function in a Deep Q-learning (DQN) formulation of combinatorial optimization on graphs, using a GNN [Dai *et al.*, 2016] to embed nodes of the input graph. Through the combination of GNN and DQN, a greedy node selection policy, e.g., iteratively selecting a subset of nodes in a graph, is learned on a set of problem instances drawn from the same distribution. In the following, we discuss extensions and improvements of the work of Khalil *et al.* [2017], categorized along three axes: The combinatorial optimization problems being addressed, the use of custom GNN architectures that improve over standard ones in some respect, and the use of specialized training approaches that alleviate bottlenecks in typical supervised or reinforcement learning (RL) for CO.

Modeling Combinatorial Problems and Handling Constraints Kool *et al.* [2019] tackle routing-type problems by training an encoder-decoder architecture, based on Graph Attention Networks [Veličković *et al.*, 2018], using an Actor-Critic reinforcement learning approach. Thus far, the problems discussed in this section have constraints that are relatively easy to satisfy, e.g., by restricting the reinforcement learning *action space* appropriately. In many practical problems, some of the constraints may be trickier to satisfy. Consider the more general TSP with Time Windows (TSPTW) [Savelsbergh, 1985], in which a node can only be visited within a node-specific time window. Here, edge weights should be interpreted as travel times rather than distances. Ma *et al.* [2019] tackle the TSPTW by augmenting the building blocks we have discussed so far (GNN with reinforcement learning) with a *hierarchical* perspective. Some of the learnable parameters are responsible for generating feasible solutions, while others minimize the solution cost. Note, however, that the approach of Ma *et al.* [2019] may still produce infeasible solutions, although it is reported to do so very rarely in experiments. Liu *et al.* [2021] employ GNNs to lean chordal extensions in graphs. Specifically, they employ an on-policy imitation learning approach to imitate

the minimum degree heuristic. For SAT problems, Selsam *et al.* [2019] introduce the NeuroSAT architecture, a GNN that learns to solve SAT instances in an end-to-end fashion by modelling them as bipartite graphs. The model is directly trained to act as a satisfiability classifier, which was further investigated in [Cameron *et al.*, 2020], also showing that GNNs are capable of generalizing to larger random instances.

GNN Architectures Deudon *et al.* [2018], Nazari *et al.* [2018], and Kool *et al.* [2019] were perhaps the first to use attention-based models for routing problems. As one moves from basic problems to richer ones, the GNN architecture's flexibility becomes more important in that it should be easy to incorporate additional characteristics of the problem. Notably, the encoder-decoder model of [Kool *et al.*, 2019] is adjusted for each type of problem to accommodate its special characteristics, e.g., node penalties and capacities, the constraint that a feasible tour must include all nodes or the lack thereof, et cetera. This allows for a unified learning approach that can produce good heuristics for different optimization problems. Recently, Joshi *et al.* [2019] propose the use of residual gated graph convolutional networks [Bresson and Laurent, 2017] in a supervised manner to solve the TSP. Unlike most previous approaches, the model does not output a valid TSP tour but a probability for each edge to belong to the tour. The final circuit is computed subsequently using a greedy decoding or a beam-search procedure. The current limitations of GNN architectures for finding good primal solutions have been analyzed in [Joshi *et al.*, 2020], using the TSP as a case study. Besides, François *et al.* [2019] have shown that the solutions obtained by Deudon *et al.*; Kool *et al.*; Joshi *et al.*; Khalil *et al.* [2018; 2019; 2019; 2017] can be efficiently used as the first solution of a local search procedure for solving the TSP.

[Li *et al.*, 2019; Fey *et al.*, 2020] investigate using GNNs for graph matching. Here, graph matching refers to finding an alignment between two graphs such that a cost function is minimized, i.e., similar nodes in one graph are matched to similar nodes in the other graph. Specifically, Li *et al.* [2019] use a GNN architecture that learns node embeddings for each node in the two graphs and an attention score that reflects the similarity between two nodes across the two graphs. The authors propose to use pair-wise and triplet losses to train the above architecture. Fey *et al.* [2020] propose a two-stage architecture for the above matching problem. In the first stage, a GNN learns a node embedding to compute a similarity score between nodes based on local neighborhoods. To fix potential miss-alignments due to the first stage's purely local nature, the authors propose a differentiable, iterative refinement strategy that aims to reach a consensus of matched nodes.

Training Approaches Toenshoff *et al.* [2019] propose a purely unsupervised approach for solving constrained optimization problems on graphs. Thereto, they trained a GNN on the bipartite constraint-variable graph using an unsupervised loss function, reflecting how the current solution adhers the constraints. Further, Karalias and Loukas [2020] propose an unsupervised approach with theoretical guarantees. Concretely, they use a GNN to produce a distribution over subsets of nodes, representing possible solution of the given problem, by minimizing a probabilistic penalty loss function. To arrive at an integral solution, they de-randomize the continuous values, using sequential decoding, and show that this integral solution obeys the given, problem-specific constraints with high probability. Nowak *et al.* [2018] train a GNNs in a supervised fashion to predict solutions to the Quadratic Assignment Problem (QAP). To do so, they represent QAP instances as two adjacency matrices, and use the two corresponding graphs as input to a GNN. Prates *et al.* [2019] train a GNN in a supervised manner to predict the satisfiability of the decision version of the TSP, considering instances up to 105 cities.

Learning Hybrid Heuristics

Within the RL framework for learning heuristics for graph problems, [Abe *et al.*, 2019] propose to guide a Monte-Carlo Tree Search (MCTS) algorithm using a GNN, inspired by the success of AlphaGo Zero [Silver *et al.*, 2017]. A similar approach appears in [Drori *et al.*, 2020]. Despite the popularity of the RL approach for CO heuristics, Li *et al.* [2018] propose a supervised learning framework which, when coupled at test time with classical algorithms such as tree search and local search, performs favorably when compared to S2V-DQN and non-learned heuristics. They use Graph Convolutional Networks [Kipf and Welling, 2017, GCNs], a simple GNN architecture, on combinatorial problems that are easy to reduce to the Maximum Independent Set (MIS), again a problem on a graph. A training instance is associated with a label, i.e., an optimal solution. The GCN is then trained to output *multiple* continuous solution predictions, and the hindsight loss function [Guzman-Rivera *et al.*, 2012] considers the minimum (cross-entropy) loss value across the multiple predictions. As such, the training encourages the generation of diverse solutions. At test time, these multiple (continuous) predictions are passed on to a tree search and local search in an attempt to transform them into feasible, potentially high-quality solutions.

Nair *et al.* [2020] propose a neighborhood search heuristic for ILPs called neural diving as a two-step procedure. By using the bipartite graph induced by the variable constraint relationship, they first train a GNN by energy modeling to predict feasible assignments, with higher probability given to better objective values. The GNN is used to produce a tentative assignment of values, and in a second step, some of these values are thrown away, then computed again by an integer programming solver by solving the sub-ILP obtained by fixing the values of those variables that were kept. A binary classifier is trained to predict which variables should be thrown away at the second step.

Ding *et al.* [2020] explore to leverage GNNs to approximately solve MIPs by representing them as a tripartite graph consisting of variable and constraint nodes, and a single objective node. Here, a variable and constraint node share an edge if the variable participates in the constraints with a non-zero coefficient. The objective shares an edge with every other node. The GNN aims to predict if a binary variable should be assigned 0 or 1. They utilize the output, i.e., a variable assignment for binary variables, of the GNN to generate either local branching global cuts [Fischetti and Lodi, 2003] or using these cuts to branch at the root node. Since the generation of labeled training data is costly, they resort to predicting so-called *stable variables*, i.e., a variable whose assignment does not change over a given set of feasible solutions.

Concerning SAT problems, Yolcu and Póczos [2019] propose to encode SAT instances as an edge-labeled, bipartite graph and used a reinforcement learning approach, namely REINFORCE parameterized by a GNN, to learn satisfying assignments inside a

stochastic local search procedure, representing each clause and variable as a node. Here, a clause and a variable share an edge if the variable appears in the clause, while the edge labels indicate if a variables is negated in the corresponding clause.

2.2 On the Dual Side: Proving Optimality

Besides finding solutions that achieve as good an objective value as possible, another common task in CO is proving that a given solution is optimal, or at least proving that the gap between the best found objective value and the optimal objective value, known as the *optimality gap*, is no greater than some bound. Computing such bound is usually achieved by computing cheap relaxations of the optimization problem. A few works have successfully used GNNs to guide or enhance algorithms to achieve this goal. Because the task's objective is to offer proofs (of optimality or the validity of a bound), GNNs usually replace specific algorithms' components.

In integer linear programming, the prototypical algorithm is branch and bound, forming the core of all state-of-the-art solving software. Here, branching attempts to bound the optimality gap and eventually prove optimality by recursively dividing the feasible set and computing relaxations to prune away subsets that cannot contain the optimal solution. An arbitrary choice usually has to be made to divide a subset by choosing a variable whose range will be divided in two. As this choice has a significant impact on the algorithm's execution time, there has been increased interest in learning policies, e.g., parameterized by a GNN, to select the best variable in a given context. The first such work was the approach of Gasse *et al.* [2019], who teach a GNN to imitate strong branching, an expert policy taking excellent decisions, but computationally too expensive to use in practice. The resulting policy leads to faster solving times than the solver default procedure and generalizes to larger instances than trained on. Building on that, Gupta *et al.* [2020] propose a hybrid branching model using a GNN at the initial decision points and a light multilayer perceptron for subsequent steps, showing improvements on CPU-only hardware. Also, Sun *et al.* [2020] uses a GNN learned with evolution strategies to improve on the GNN of Gasse *et al.* [2019] on problems defined on graphs sharing a common backbone. Finally, Nair *et al.* [2020] expand the GNN approach to branching by implementing a GPU-friendly parallel linear programming solver using the alternating direction method of multipliers that allows scaling the strong branching expert to substantially larger instances. Combining this innovation with a novel GNN approach to primal diving (see Section 2.1) they show improvements over SCIP [Gamrath *et al.*, 2020] in solving time on five real-life benchmarks and MIPLIB [Gleixner *et al.*, 2020], a standard benchmark of heterogeneous instances.

In logic solving, such as for Boolean Satisfiability, Satisfiability Modulo Theories, and Quantified Boolean Formulas solving, a standard algorithm is Conflict-Driven Clause Learning (CDCL). CDCL is a backtracking search algorithm that resolves conflicts with resolution steps. In this algorithm, one must repeatedly branch, i.e., pick an unassigned variable and a polarity (value, 0 or 1) to assign to this variable. Some authors have proposed representing logical formulas as graphs and using a GNN to select the best next variable, the analog of a branching step. Namely, Lederman *et al.* [2020] model quantified boolean formulas as bipartite graphs and teach a GNN to branch using REINFORCE, achieving substantial improvements in number of formulas solved within a given time limit compared to VSIDS, the standard branching heuristic. Kurin *et al.*; **?** [2020; **?**] explore similar ideas using Q-learning and evolution strategies, respectively. Finally, in a different direction, Selsam and Bjørner [2019] use the end-to-end GNN NeuroSAT architecture [Selsam *et al.*, 2019], a GNN on the bipartite variable-clause graph, inside existing SAT solvers, e.g., MiniSat, Glucose, and Z3, to inform variable branching decisions. They train the GNN to predict the probability of a variable being in an unfeasible core, and assume that this probability correlates well with being a good variable to branch on. Using the resulting network for branching periodically, they report solving more problems on standard benchmarks than the state-of-the-art heuristic EVSIDS.

In constraint programming, optimal solutions are found using backtracking search algorithms, e.g., branch and bound or iterative limited discrepancy search, repeatedly selecting variables and corresponding value assignments, similarly to logic solvers. Value selection has, in particular, a significant impact on the quality of the search. In the case of constraint satisfaction or optimization programs that can be formulated as Markov decision processes on graph states, such as the TSP with time windows, Cappart *et al.* [2020] train a GNN to learn a good policy or action-value function using reinforcement learning to drive value selection within the backtracking search algorithms of CP solvers. This idea has been further extended by Chalumeau *et al.* [2021], who propose a new CP solver that natively handles a learning component.

Finally, a recently introduced, generic way of obtaining dual bounds in CO problems is through *decision diagrams* [Bergman *et al.*, 2016]. These are graphs that can be used to encode the feasible space of discrete problems. For some of those problems, it is possible to identify an appropriate *merging operator* that yields relaxed decision diagrams, whose best solution gives a dual bound. However, the bound's quality is highly dependent on the variable ordering that has been considered to construct the diagram. Cappart *et al.* [2019] train a GNN by reinforcement learning to decide which variable to add next to an incomplete decision diagram representing the problem instance that needs to be solved. The resulting diagram then readily yields a bound on the optimal objective value of the problem.

2.3 Algorithmic Reasoning

Neural networks are traditionally powerful in the *interpolation* regime, i.e., when we expect the distribution of unseen ("test") inputs to roughly match the distribution of the inputs used to train the network. However, they tend to struggle when *extrapolating*, i.e., when they are evaluated out of distribution. Extrapolating is a potentially important issue for tackling CO problems with (G)NNs trained end to end. As a critical feature of a powerful reasoning system, it should apply to *any* plausible input, not just those within the training distribution. Therefore, unless we can foreshadow the kinds of inputs our neural CO approach will be solving, it could be essential to address out-of-distribution generalization meaningfully.

One resurging research direction that holds much promise here is algorithmic reasoning, i.e., directly introducing concepts from classical algorithms [Cormen *et al.*, 2009] into neural network architectures or training regimes, typically by learning how to

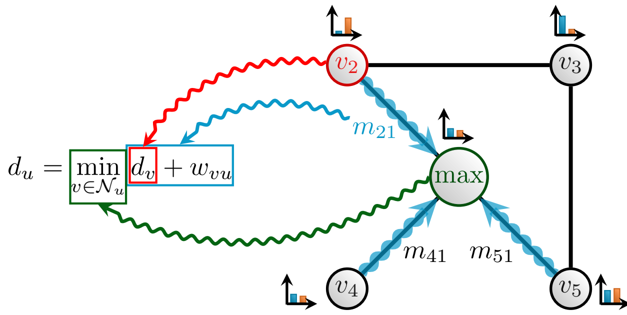

Figure 1: Algorithmic alignment, in the case of the Bellman-Ford shortest path-finding algorithm. It computes distance estimates for every node, d_u. Specifically, a GNN aligns well with this dynamic programming update. Node features align with intermediate computed values (**red**), message functions align with the candidate solutions from each neighbor (**blue**), and the aggregation function aligns with the optimization across neighbors (**green**).

execute them. Classical algorithms have precisely the kind of favorable properties (strong generalization, compositionality, verifiable correctness) that would be desirable for neural network reasoners. Bringing the two sides closer together can therefore yield the kinds of improvements to performance, generalization, and interpretability that are unlikely to occur through architectural gains alone. Further, initial theoretical analyses [Xu *et al.*, 2019b] demonstrated that GNNs align with dynamic programming [Bellman, 1966] (Figure 1), which is a language in which most algorithms can be expressed. This motivates the use of GNNs in this setting.

Algorithmic Alignment The concept of algorithmic alignment introduced in [Xu *et al.*, 2019b] is central to constructing effective algorithmic reasoners that extrapolate better. Informally, a neural network *aligns* with an algorithm if that algorithm can be partitioned into several parts, each of which can be "easily" modeled by one of the neural network's modules. Essentially, alignment relies on designing neural networks' components and control flow such that they line up well with the underlying algorithm to be learned from data. The work from [Veličković *et al.*, 2020b] on neural execution of graph algorithms is among the first to propose algorithmic learning as a first-class citizen and suggests several general-purpose modifications to GNNs to make them stronger combinatorial reasoners. These include using the encode-process-decode paradigm [Hamrick *et al.*, 2018], favoring $\max$-aggregation, or leveraging strong supervision [Palm *et al.*, 2017].

It should be noted that, concurrently, significant strides have been made on using GNNs for physics simulations [Sanchez-Gonzalez *et al.*, 2020; Pfaff *et al.*, 2020], coming up with a largely equivalent set of prescriptions. Simulations and algorithms can be seen as two sides of the same coin. Algorithms can be phrased as discrete-time simulations, and, as physical hardware cannot support a continuum of inputs, simulations are typically realized as step-wise algorithms. As such, the observed correspondence comes as little surprise—any progress made in neural algorithmic reasoning is likely to translate into progress for neural physical simulations and vice-versa. Several works have expanded on these prescriptions even further, yielding stronger classes of GNN executors. IterGNNs [Tang *et al.*, 2020] provably align well with *iterative algorithms*, adaptively learning a stopping criterion without requiring an explicit termination network. Neural Shuffle-exchange Networks [Freivalds *et al.*, 2019; Draguns *et al.*, 2020] directly fix connectivity patterns between nodes based on results from routing theory, allowing them to efficiently align with $O(n \log n)$ sequence processing algorithms. Lastly, pointer graph networks (PGNs) [Veličković *et al.*, 2020a] take a more pragmatic view of this issue. The graph used by the processor GNN needs not to match the input graph, which may not even be given in many problems of interest. Instead, PGNs explicitly predict a graph to be used by the processor, enforcing it to match data structures' behavior.

Lastly, recent theoretical results have provided a unifying explanation for why algorithmically inspired GNNs provide benefits to extrapolating both in algorithmic and in physics-based tasks. Specifically, [Xu *et al.*, 2020] make a useful geometric argument—ReLU MLPs tend to extrapolate *linearly* outside of the training set support. That is, learning roughly-linear ground-truth functions, e.g., by message functions in GNNs, implies stronger out-of-distribution performance.

Reasoning on Natural Inputs Classical algorithms are designed with abstraction in mind, enforcing their inputs to conform to stringent preconditions. This is done for an apparent reason, keeping the inputs constrained enables an uninterrupted focus on "reasoning" and makes it far easier to certify the resulting procedure's correctness, i.e., stringent postconditions. However, we must never forget why we design algorithms, to apply them to real-world problems.

Being able to satisfy algorithms' preconditions necessitates converting their inputs into an abstractified form, which, if done manually, often implies drastic information loss, meaning that our combinatorial problem no longer accurately portrays the dynamics of the real world. The data we need to apply the algorithm may also be only partially observable, and this can often render the algorithm completely inapplicable. Both points should be recognized as important issues within the combinatorial optimization as well as operations research communities. Further, they present fertile ground for neural networks. However, even if we use a neural network to encode inputs for a classical combinatorial algorithm properly, due to the discrete nature of CO problems, usual gradient-based computation is often not applicable.

Although promising ways to tackle the issue of gradient estimation have already emerged in the literature [Knöbelreiter *et al.*, 2017; Wang *et al.*, 2019; Vlastelica *et al.*, 2020; Mandi and Guns, 2020], another critical issue to consider is data efficiency. Even if a feasible backward pass becomes available for a combinatorial algorithm, the potential richness of raw data still needs to be bottlenecked to a scalar value. While explicitly recovering such a value allows for easier interpretability of the system, the solver is still committing to using it; its preconditions often assume inputs are free of noise and estimated correctly. In contrast, neural networks derive flexibility from their latent representations, that are inherently high-dimensional; if any component of the neural representation ends up poorly predicted, other components are still able to step in and compensate.

Mindful of the above, advances in neural algorithmic reasoning

could lend a remarkably elegant pipeline for reasoning on natural inputs. The power comes from using the aforementioned encode-process-decode framework [Hamrick *et al.*, 2018]. Assume we have trained a GNN executor to perform a target algorithm on many (synthetically generated) abstract inputs. The executor trained as prescribed before will have a processor network P, which directly emulates one step of the algorithm, in the latent space. Thus, within the weights of a properly-trained processor network, we find a combinatorial algorithm that is

(a) aligned with the computations of the target algorithm;
(b) operates by matrix multiplications, hence natively admits useful gradients;
(c) operates over high-dimensional latent spaces, hence is not vulnerable to bottleneck phenomena and may be more data-efficient.

The caveat is that, being a tractable-depth GNN, the computations of P are necessarily in P. Unless P=NP, this is unlikely to be enough to produce general solutions to NP-hard problems (only their polynomial-time heuristics).

The general procedure for applying an algorithm A (which admits abstract inputs $\bar{x}$) to raw inputs x is as follows:

1. Learn an algorithmic reasoner for A, on synthetic inputs, $\bar{x}$, using the encode-process-decode pipeline. This yields functions f, P, g such that $g(P(f(\bar{x}))) \approx A(\bar{x})$.
2. Design encoder and decoder neural networks, $\tilde{f}$ and $\tilde{g}$, to process raw data and produce desirable outputs. $\tilde{f}$ should produce embeddings that correspond to the input dimension of P, while $\tilde{g}$ should operate over input embeddings that correspond to the output dimension of P.
3. Learn parameters of $\tilde{f}$ and $\tilde{g}$ by gradient descent on any differentiable loss function that compares $\tilde{g}(P(\tilde{f}(x)))$ to ground-truth outputs, y. Parameters of P should be frozen.

Therefore, algorithmic reasoning presents a strong approach—through pre-trained processors—to reasoning over natural inputs. The raw encoder function, $\tilde{f}$, has the potential to further enhance the CO pipeline, as is learning how to map raw inputs onto the algorithmic input space for P, purely by backpropagation. This construction has already yielded useful architectures in the space of reinforcement learning, mainly implicit planning. Herein, the XLVIN architecture [Deac *et al.*, 2020] has enabled the value iteration algorithm to be executed on arbitrary reinforcement environments by a direct application of this blueprint.

3 Future Research Directions

In the following, we propose several directions for stimulating future research.

Understanding the Trade-off in Scalability, Expressivity, and Generalization Current GNN architectures might miss crucial structural patterns in the data [Morris *et al.*, 2019; Xu *et al.*, 2019a], while more expressive approaches [Morris *et al.*, 2020; Maron *et al.*, 2019] do not scale to large-scale inputs. What is more, decisions inside combinatorial optimization solvers, e.g., a branching decision, are often driven by simple heuristics that are cheap to compute. Although negligible when called only a few times, resorting to a GNN inside a solver for such decisions is time-consuming compared to a simple heuristic. Moreover, internal computations inside a solver can hardly be parallelized. Hence, devising GNN architectures that scale and simultaneously capture essential pattern remains an open challenge. However, increased expressiveness might negatively impact generalization. Hence, understanding the trade-off between these three aspects remains an open challenge for deploying GNNs on combinatorial tasks.

Relying on a Limited Number of Data and the Use of Reinforcement Learning The final goal of machine learning-based CO solvers is to leverage knowledge from previously solved instances to solve future ones better. Many works in this survey hypothesize that an infinite amount of data is available for this purpose. However, unlimited labeled training is not available in practice. Further, in many cases, it may be challenging to procure labeled data. Hence, an open challenge is to develop approaches able to learn efficiently with a restricted number of potentially unlabeled instances. An obvious candidate circumventing the need for labeled training data is reinforcement learning. Compared to supervised approaches the systematic use of reinforcement learning to solve CO problems is only at the beginning, which is most likely because these approaches are hard to train, and there is little understanding of which reinforcement learning approaches are suitable for CO problems. Hence, adapting currently used RL agents to CO problems' specific needs remains another key challenge.

Programmatic Primitives While existing work in algorithmic reasoning already can use GNNs to align with data structure-backed iterative algorithms comfortably, there exist many domains and constructs that are of high interest to CO but are still not explicitly treated by this emerging area. As only a few examples, we highlight string algorithms, (common in bioinformatics), flow algorithms, and explicitly supporting recursive primitives, for which any existing GNN executor would eventually run out of representational capacity.

Perceptive CO Significant strides were already made to use GNNs to strengthen abstractified CO pipelines. Further efforts are needed to support combinatorial reasoning over real-world inputs as CO problems are often designed as proxies for solving them. Our algorithmic reasoning section hints at a few possible blueprints for supporting this, but all of them are still in the early stages. One issue still untackled by prior research is how to meaningfully extract variables for the CO optimizer when they are not trivially given. While natural inputs pose several such challenges for the CO pipeline, it is equally important to keep in mind that "nature is not an adversary"—even if the underlying problem is NP-hard, the instances provided in practice may well be effectively solvable with fast heuristics, or, in some case exactly.

4 Conclusions

We gave an overview of the recent applications of GNNs for CO. To that, we surveyed primal approaches that aim at finding a heuristic or optimal solution with the help of GNNs. We then explored recent dual approaches, i.e., ones that use GNNs to facilitate proving that a given solution is optimal. Moreover, we gave an overview of algorithmic reasoning, i.e., data-driven approaches aiming to overcome classical algorithms' limitations. Finally, we identified a set of critical challenges to stimulate future research and advance the emerging field.

References

[Abe *et al.*, 2019] K. Abe, I. Sato, and M. Sugiyama. Solving NP-hard problems on graphs by reinforcement learning without domain knowledge. *Simulation*, 1:1–1, 2019.

[Bellman, 1966] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[Bengio *et al.*, 2021] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *EJOR*, 290(2):405–421, 2021.

[Bergman *et al.*, 2016] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[Boussaïd *et al.*, 2013] I. Boussaïd, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[Bresson and Laurent, 2017] X. Bresson and T. Laurent. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017.

[Cameron *et al.*, 2020] C. Cameron, R. Chen, J. S. Hartford, and K. Leyton-Brown. Predicting propositional satisfiability via end-to-end learning. In *AAAI*, pages 3324–3331, 2020.

[Cappart *et al.*, 2019] Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *AAAI*, pages 1443–1451, 2019.

[Cappart *et al.*, 2020] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz, and A. Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. *CoRR*, abs/2006.01610, 2020.

[Cappart *et al.*, 2021] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Velickovic. Combinatorial optimization and reasoning with graph neural networks. *CoRR*, abs/2102.09544, 2021.

[Chalumeau *et al.*, 2021] F. Chalumeau, I. Coulon, Q. Cappart, and L.-M. Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. *CoRR*, abs/2102.09193, 2021.

[Cormen *et al.*, 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2009.

[Dai *et al.*, 2016] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *ICML*, pages 2702–2711, 2016.

[Deac *et al.*, 2020] A. Deac, P. Veličković, O Milinković, P.-L. Bacon, J. Tang, and M. Nikolić. XLVIN: eXecuted Latent Value Iteration Nets. *CoRR*, abs/2010.13146, 2020.

[Deudon *et al.*, 2018] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the tsp by policy gradient. In *CPAIOR*, pages 170–181, 2018.

[Ding *et al.*, 2020] J.-Y. Ding, C. Zhang, L. Shen, S. Li, B. Wang, Y. Xu, and L. Song. Accelerating primal solution findings for mixed integer programs based on solution prediction. In *AAAI*, 2020.

[Draguns *et al.*, 2020] A. Draguns, E. Ozolins, A. Sostaks, M. Apinis, and K. Freivalds. Residual shuffle-exchange networks for fast processing of long sequences. *CoRR*, abs/2004.04662, 2020.

[Drori *et al.*, 2020] I. Drori, A. Kharkar, W. R. Sickinger, B. Kates, Q. Ma, S. Ge, E. Dolev, B. Dietrich, D. P. Williamson, and M. Udell. Learning to solve combinatorial optimization problems on real-world graphs in linear time. *CoRR*, abs/2006.03750, 2020.

[Fey *et al.*, 2020] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege. Deep graph matching consensus. In *ICLR*, 2020.

[Fischetti and Lodi, 2003] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98(1-3):23–47, 2003.

[François *et al.*, 2019] A. François, Q. Cappart, and L.-M. Rousseau. How to evaluate machine learning approaches for combinatorial optimization: Application to the travelling salesman problem. *CoRR*, abs/1909.13121, 2019.

[Freivalds *et al.*, 2019] K. Freivalds, E. Ozoliņš, and Šostaks. Neural shuffle-exchange networks-sequence processing in O(n log n) time. In *NeurIPS*, pages 6626–6637, 2019.

[Gamrath *et al.*, 2020] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig. The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, March 2020.

[Gasse *et al.*, 2019] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *NeurIPS*, pages 15554–15566, 2019.

[Gilmer *et al.*, 2017] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

[Gleixner *et al.*, 2020] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 2020.

[Gupta *et al.*, 2020] P. Gupta, M. Gasse, E. B. Khalil, M. P. Kumar, A. Lodi, and Y. Bengio. Hybrid models for learning to branch. *CoRR*, abs/2006.15212, 2020.

[Guzman-Rivera *et al.*, 2012] A. Guzman-Rivera, D. Batra, and P. Kohli. Multiple choice learning: Learning to produce multiple structured outputs. In *NeurIPS*, pages 1808–1816, 2012.

[Hamrick *et al.*, 2018] J. B. Hamrick, K. R. Allen, V. Bapst, T. Zhu, K. R. McKee, J. B. Tenenbaum, and P. W. Battaglia. Relational inductive bias for physical construction in humans and machines. In *Annual Meeting of the Cognitive Science Society*, 2018.

[Joshi *et al.*, 2019] C. K. Joshi, T. Laurent, and X. Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *CoRR*, abs/1906.01227, 2019.

[Joshi *et al.*, 2020] C. K. Joshi, Q. Cappart, L.-M. Rousseau, T. Laurent, and X. Bresson. Learning TSP requires rethinking generalization. *CoRR*, abs/2006.07054, 2020.

[Karalias and Loukas, 2020] N. Karalias and A. Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. In *NeurIPS*, 2020.

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

[Khalil *et al.*, 2017] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *NeurIPS*, pages 6348–6358, 2017.

[Kipf and Welling, 2017] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[Knöbelreiter *et al.*, 2017] P. Knöbelreiter, C. Reinbacher, A. Shekhovtsov, and T. Pock. End-to-end training of hybrid cnn-crf models for stereo. In *CVPR*, pages 2339–2348, 2017.

[Kool *et al.*, 2019] W. Kool, H. Van Hoof, and M. Welling. Attention, learn to solve routing problems! In *ICLR*, 2019.

[Korte and Vygen, 2012] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 5th edition, 2012.

[Kotary *et al.*, 2021] J. Kotary, F. Fioretto, P. Van Hentenryck, and B. Wilder. End-to-end constrained optimization learning: A survey. *CoRR*, abs/2103.16378, 2021.

[Kurin *et al.*, 2020] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro. Can Q-learning with graph networks learn a generalizable branching heuristic for a SAT solver? In *NeurIPS*, 2020.

[Lederman *et al.*, 2020] G. Lederman, M. N. Rabe, and S. A. Seshia. Learning heuristics for automated reasoning through deep reinforcement learning. In *ICLR*, 2020.

[Li *et al.*, 2018] Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *NeurIPS*, pages 537–546, 2018.

[Li *et al.*, 2019] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph matching networks for learning the similarity of graph structured objects. In *ICML*, pages 3835–3845, 2019.

[Liu *et al.*, 2021] D. Liu, A. Lodi, and M. Tanneau. Learning chordal extensions. *Journal of Global Optimization*, 2021.

[Ma *et al.*, 2019] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *CoRR*, abs/1911.04936, 2019.

[Mandi and Guns, 2020] J. Mandi and T. Guns. Interior point solving for lp-based prediction+optimisation. In *Advances in Neural Information Processing Systems*, 2020.

[Maron *et al.*, 2019] H. Maron, H. Ben-Hamu, N. Shamir, and Y. Lipman. Invariant and equivariant graph networks. In *ICLR*, 2019.

[Morris *et al.*, 2019] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *AAAI*, pages 4602–4609, 2019.

[Morris *et al.*, 2020] C. Morris, G. Rattan, and P. Mutzel. Weisfeiler and Leman go sparse: Towards higher-order graph embeddings. In *NeurIPS*, 2020.

[Nair *et al.*, 2020] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandrtmadja, P. Wang, et al. Solving mixed integer programs using neural networks. *CoRR*, abs/2012.13349, 2020.

[Nazari *et al.*, 2018] M. Nazari, A. Oroojlooy, M. Takáč, and L. V. Snyder. Reinforcement learning for solving the vehicle routing problem. In *NeurIPS*, pages 9861–9871, 2018.

[Nowak *et al.*, 2018] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. Revised note on learning quadratic assignment with graph neural networks. In *2018 IEEE Data Science Workshop (DSW)*, pages 1–5, 2018.

[Palm *et al.*, 2017] R. B. Palm, U. Paquet, and O. Winther. Recurrent relational networks. *Corr*, abs/1711.08028, 2017.

[Pfaff *et al.*, 2020] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning mesh-based simulation with graph networks. *CoRR*, abs/2010.03409, 2020.

[Prates *et al.*, 2019] M. O. R. Prates, P. H. C. Avelar, H. Lemos, L. C. Lamb, and M. Y. Vardi. Learning to solve NP-complete problems: A graph neural network for decision TSP. In *AAAI*, pages 4731–4738, 2019.

[Sanchez-Gonzalez *et al.*, 2020] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia. Learning to simulate complex physics with graph networks. *CoRR*, abs/2002.09405, 2020.

[Savelsbergh, 1985] M. W. P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

[Selsam and Bjørner, 2019] D. Selsam and N. Bjørner. NeuroCore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR*, abs/1903.04671, 2019.

[Selsam *et al.*, 2019] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a SAT solver from single-bit supervision. In *ICLR*, 2019.

[Silver *et al.*, 2017] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[Sun *et al.*, 2020] H. Sun, W. Chen, H. Li, and Le Song. Improving learning to branch via reinforcement learning. In *Workshop on Learning Meets Combinatorial Algorithms, NeurIPS*, 2020.

[Tang *et al.*, 2020] H. Tang, Z. Huang, J. Gu, B.-L. Lu, and H. Su. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *NeurIPS*, 33, 2020.

[Toenshoff *et al.*, 2019] J. Toenshoff, M. Ritzert, H. Wolf, and M. Grohe. RUN-CSP: unsupervised learning of message passing networks for binary constraint satisfaction problems. *CoRR*, abs/1909.08387, 2019.

[Vazirani, 2010] V. V. Vazirani. *Approximation Algorithms*. Springer, 2010.

[Veličković *et al.*, 2018] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[Veličković *et al.*, 2020a] P. Veličković, L. Buesing, M. C. Overlan, R. Pascanu, O. Vinyals, and C. Blundell. Pointer graph networks. *CoRR*, abs/2006.06380, 2020.

[Veličković *et al.*, 2020b] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. Neural execution of graph algorithms. In *ICLR*, 2020.

[Vlastelica *et al.*, 2020] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek. Differentiation of blackbox combinatorial solvers. In *ICLR*, 2020.

[Wang *et al.*, 2019] P.-W. Wang, P. Donti, B. Wilder, and Z. Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *ICML*, pages 6545–6554, 2019.

[Xu *et al.*, 2019a] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[Xu *et al.*, 2019b] K. Xu, J. Li, M. Zhang, S. S. Du, K. Kawarabayashi, and S. Jegelka. What can neural networks reason about? *CoRR*, abs/1905.13211, 2019.

[Xu *et al.*, 2020] K. Xu, J. Li, M. Zhang, S. S. Du, K. Kawarabayashi, and S. Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. *CoRR*, abs/2009.11848, 2020.

[Yolcu and Póczos, 2019] R. Yolcu and B. Póczos. Learning local search heuristics for boolean satisfiability. In *NeurIPS*, pages 7992–8003, 2019.