Combinatorial Optimization and Reasoning with Graph Neural Networks

Quentin Cappart QUENTIN.CAPPART@POLYMTL.CA
Department of Computer Engineering and Software Engineering
Polytechnique Montréal
Montréal, Canada

Didier Chételat DIDIER.CHETELAT@POLYMTL.CA
CERC in Data Science for Real-Time Decision-Making
Polytechnique Montréal
Montréal, Canada

Elias B. Khalil KHALIL@MIE.UTORONTO.CA
Department of Mechanical & Industrial Engineering,
University of Toronto
Toronto, Canada

Andrea Lodi ANDREA.LODI@POLYMTL.CA
CERC in Data Science for Real-Time Decision-Making
Polytechnique Montréal
Montréal, Canada

Christopher Morris CHRIS@CHRISTOPHERMORRIS.INFO
CERC in Data Science for Real-Time Decision-Making
Polytechnique Montréal
Montréal, Canada

Petar Veličković PETARV@GOOGLE.COM
DeepMind
London, UK

Abstract

Combinatorial optimization is a well-established area in operations research and computer science. Until recently, its methods have 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 (GNNs), as a key building block for combinatorial tasks, either directly as solvers or by enhancing exact solvers. The inductive bias of GNNs effectively encodes combinatorial and relational input due to their invariance to permutations and awareness of input sparsity. This paper presents a conceptual review of recent key advancements in this emerging field, aiming at researchers in both optimization and machine learning.

Keywords: Combinatorial optimization, graph neural networks, reasoning

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; see (Korte and Vygen,

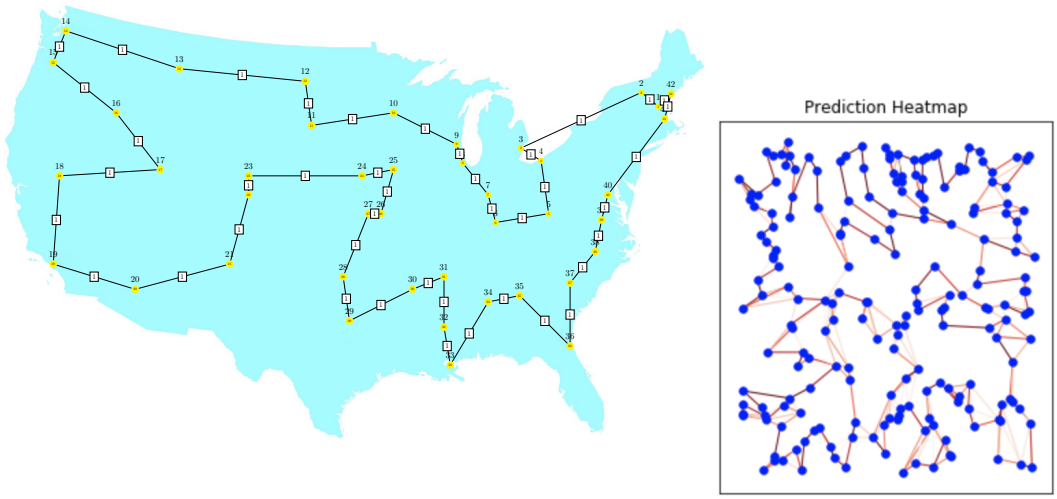


Figure 1: Routing problems are naturally framed in the language of combinatorial optimization. For example, the problem of optimally traversing the 48 contiguous US states (visiting one city per state) can be expressed as the *Travelling Salesperson Problem* (TSP). An optimal solution to this problem instance (left) was found by Dantzig et al. (1954). Graphs are a useful representation of the transportation networks underlying TSP instances, and so GNNs have been used successfully to search for approximate solutions (right; reprinted from (Joshi et al., 2020)).

2012) for a general overview. 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. For example, a trucking company may solve TSP instances for the same city on a daily basis, with only slight differences across instance in the travel times due to varying traffic conditions. 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). 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. For example, well-known and relevant problems such as the Traveling Salesperson problem and other vehicle routing problems naturally induce a graph structure (Figure 1). In fact, from the 21 NP-complete problems identified by Karp (1972), ten are decision versions of graph optimization problems. 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.

1.1 What are the Challenges for Machine Learning?

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. 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. Further, the machine learning method has to be able to generalize beyond its training data, e.g., transferring to instances of different sizes.

Overall, there is a trade-off between scalability, expressivity, and generalization, any pair of which might conflict. In summary, the key challenges are:

1. Machine learning methods that operate on graph data have to be *invariant* to node *permutations*. They should also exploit the graph *sparsity*.
2. Models should *distinguish* critical structural *patterns* in the provided data while still *scaling* to large real-world instances.
3. *Side information* in the form of high-dimensional vectors attached to nodes and edges, i.e., modeling objectives and additional information, needs to be considered.
4. Models should be *data-efficient*. That is, they should ideally work without requiring large amounts of labeled data, and they should be transferable to *out-of-sample* instances.

1.2 How Do GNNs Address These Challenges?

GNNs (Gilmer et al., 2017; Scarselli et al., 2009) have recently emerged as machine learning architectures that (partially) address the challenges above.

The key idea underlying GNNs is to 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, using (stochastic) first-order optimization techniques to adapt to the given data distribution. The promise here is that the learned vector representation encodes crucial graph structures that help solve a CO problem more efficiently. GNNs are invariant

and equivariant by design, i.e., they automatically exploit the invariances or symmetries inherent to the given instance or data distribution. Due to their local nature, by aggregating neighborhood information, GNNs naturally exploit sparsity, leading to more scalable models on sparse inputs. Moreover, although scalability 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 (which we will also explore and outline), they have already proven to be useful in the context of CO. In fact, they have already been applied in various settings, either to directly predict a solution or as an integrated component of an existing solver. We will extensively investigate both of these aspects within our survey.

Perhaps one of the most widely publicized applications of GNNs in CO at the time of writing is the work by Mirhoseini et al. (2020), which studies *chip placement*. The aim is to map the nodes of a *netlist* (the graph describing the desired chip) onto a *chip canvas* (a bounded 2D space), optimizing the final power, performance, and area. The authors observe this as a combinatorial problem and tackle it using reinforcement learning. Owing to the graph structure of the netlist, at the core of the representation learning pipeline is a GNN, which computes features of netlist nodes to preserve the symmetries in the netlist. This represents the first chip placement approach that can quickly generalize to previously unseen netlists, generating optimized placements for Google's TPU accelerators (Jouppi et al., 2017). While this approach has received wide coverage in the popular press, we believe that it is only scratching the surface of the innovations that can be enabled from a careful synergy of GNNs and CO, and we have designed our survey to facilitate future research in this emerging area.

1.3 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, e.g., specifying the underlying graph itself. The discrete graph input, forming the basis of most CO problems, is seldom directly induced by the raw data, requiring costly and error-prone feature engineering. This might lead to biases that do not align with the real world and consequently imprecise decisions. Such issues have been known as early as the 1950s in the context of railway network analysis (Harris and Ross, 1955), but remained out of the spotlight of theoretical computer science, which assumes problems are abstractified to begin with.

In the long-term, machine learning approaches can further enhance the CO pipeline, from raw input processing to aiding 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, along with motivating examples, in Section 3.3.4.

1.4 Present Work

In this paper, we give an overview of recent advances in the use of GNNs in the context of CO, aiming at both the CO and the machine learning researcher. To this end, we give a rigorous introduction to CO, the various machine learning regimes, and GNNs. Most importantly, we give a comprehensive, structured overview of recent applications of GNNs in the CO context. Finally, we discuss challenges arising from the use of GNNs and future work. Our contributions can be summarized as follows:

1. We provide a complete, structured overview of the application of GNNs to the CO setting, for both heuristic and exact algorithms. Moreover, we survey recent progress in using GNN-based end-to-end algorithmic reasoners.

2. We highlight the shortcomings of GNNs in the context of CO and provide guidelines and recommendations on how to tackle them.

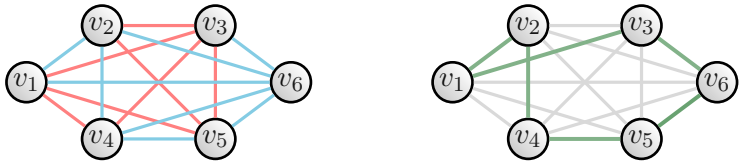
3. We provide a list of open research directions to stimulate future research.

1.5 Related Work

In the following, we briefly review key papers and survey efforts involving GNNs and machine learning for CO.

GNNs Graph neural networks (Gilmer et al., 2017; Scarselli et al., 2009) have recently (re-)emerged as the leading machine learning method for graph-structured inputs. Notable instances of this architecture include, e.g., (Duvenaud et al., 2015; Hamilton et al., 2017; Veličković et al., 2018), and the spectral approaches proposed by, e.g., Bruna et al. (2014); Defferrard et al. (2016); Kipf and Welling (2017); Monti et al. (2017)—all of which descend from early work of Kireev (1995); Sperduti and Starita (1997); Merkwirth and Lengauer (2005); Scarselli et al. (2009). Aligned with the field's recent rise in popularity, there exists a plethora of surveys on recent advances in GNN techniques. Some of the most recent ones include (Chami et al., 2020; Wu et al., 2019; Zhou et al., 2018).

Surveys The seminal survey of Smith (1999) centers around the use of popular neural network architectures of the time, namely Hopfield Networks and Self-Organizing Maps, as a basis for combinatorial heuristics. It is worth noting that such architectures were mostly used for a single instance at a time, rather than being trained over a set of training instances; this may explain their limited success at the time. Bengio et al. (2021) give a high-level overview of machine learning methods for CO, with no special focus on graph-structured input, while Lodi and Zarpellon (2017) focuses on machine learning for branching in the context of mixed-integer programming. Concurrently to our work, Kotary et al. (2021) have categorized various approaches for machine learning in CO, focusing primarily on categorizing end-to-end learning setups and paradigms, making representation learning—and GNNs in particular—a secondary topic. Moreover, the surveys by Mazyavkina et al. (2020); Yang and Whinston (2020) focus on using reinforcement learning for CO. The survey of Vesselinova et al. (2020) deals with machine learning for network problems arising in telecommunications, focusing on non-exact methods and not including recent progress. Finally, Lamb et al. (2020) gives a high-level overview of the application of GNNs in various reasoning tasks, missing out on



(a) Instance. (b) Optimal solution.

Figure 2: A complete graph with edge labels (blue and red) and its optimal solution for the TSP (in green). Blue edges have a cost of 1 and red edges a cost of 2.

the most recent developments, e.g., the algorithmic reasoning direction that we study in detail here.

1.6 Outline

We start by giving the necessary background on CO and relevant optimization frameworks, machine learning, and GNNs; see Section 2. In Section 3, we review recent research using GNNs in the context of CO. Specifically, in Section 3.1, we survey works aiming at finding primal solutions, i.e., high-quality feasible solutions to CO problems, while Section 3.2 gives an overview of works aiming at enhancing dual methods, i.e., proving the optimality of solutions. Going beyond that, Section 3.3 reviews recent research trying to facilitate algorithmic reasoning behavior in GNNs, as well as applying GNNs as raw-input combinatorial optimizers. Finally, Section 4 discusses the limits of current approaches and offers a list of research directions, stimulating future research.

2. Preliminaries

Here, we introduce notation and give the necessary formal background on (combinatorial) optimization, the different machine learning regimes, and GNNs.

2.1 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* $E \subseteq V \times V$. We denote the set of nodes and the set of edges of G by $V(G)$ and $E(G)$, respectively. A *labeled graph* G is a triplet (V, E, l) with a label function $l \colon V(G) \cup E(G) \to \Sigma$, where Σ is some finite alphabet. Then $l(x)$ is a *label* of x, for x in $V(G) \cup E(G)$. Note that x here can be either a node or an edge. The *neighborhood* of v in $V(G)$ is denoted by $N(v) = \{u \in V(G) \mid (v, u) \in E(G)\}$. A *tree* is a connected graph without cycles.

We say that two graphs G and H are *isomorphic* if there exists an edge-preserving bijection $\varphi \colon V(G) \to V(H)$, i.e., (u, v) is in $E(G)$ if and only if $(\varphi(u), \varphi(v))$ is in $E(H)$. For labeled graphs, we further require that $l(v) = l(\varphi(v))$ for v in $V(G)$ and $l((u, v)) = l((\varphi(u), \varphi(v)))$ for (u, v) in $E(G)$.

2.2 Combinatorial Optimization

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. Formally, we define an instance of a *combinatorial optimization problem* as follows.

Definition 1 (Combinatorial optimization instance) *An instance of a* combinatorial optimization problem *is a tuple* (Ω, F, w)*, where* Ω *is a* finite *set,* $F \subseteq 2^{\Omega}$ *is the set of* feasible *solutions,* $c\colon \Omega \to \mathbb{Q}$ *is a* cost function *with* $c(S) = \sum_{\omega \in S} w(\omega)$ *for* S *in* F*.*

Consequently, CO deals with selecting an element S^* (*optimal solution*) in F that minimizes $c(S^*)$ over the feasible set F.[1] The corresponding *decision problem* asks if there exists an element in the feasible set such that its cost is smaller than or equal to a given value, i.e., whether there exists S in F such that $c(S) \leq k$ (i.e., we require a Yes/No answer).

The TSP is a well-known CO problem aiming at finding a cycle along the edges of a graph with minimal cost that visits each node exactly once; see Figure 2 for an illustration of an instance of the TSP problem and its optimal solution. The corresponding decision problem asks whether there exists a cycle along the edges of a graph with cost $\leq k$ that visits each node exactly once.

Example 1 (Travelling Salesperson Problem)
Input: A complete directed graph G*, i.e.,* $E(G) = \{(u, v) \mid u, v \in V(G)\}$*, with edge costs* $w\colon E(G) \to \mathbb{Q}$*.*
Output: A permutation of the nodes $\sigma\colon \{0, \ldots, n-1\} \to V$ *such that*

$$\sum_{i=0}^{n-1} w\big((\sigma(i), \sigma((i+1) \bmod n)\big)$$

is minimal over all permutations, where $n = |V|$*.*

Due to their discrete nature, many classes or sets of combinatorial decision problems arising in practice, e.g., TSP or other vehicle routing problems, are NP-complete (Korte and Vygen, 2012), and hence likely intractable in the worst-case sense. However, instances are routinely solved in practice by formulating them as *integer linear optimization problems* or *integer linear programs* (ILPs), *constrained problems*, or as *satisfiability problems* (SAT) and utilizing well-engineered solvers for these problems, e.g., branch and cut algorithms in the case of ILPs; see the next section for details.

2.3 General Optimization Frameworks: ILPs, SAT, and Constrained Problems

In the following, we describe common CO frameworks.

2.3.1 Integer linear programs and mixed-integer programs

First, we start by defining a *linear program* or *linear optimization problem*. A linear program aims at optimizing a linear cost function over a feasible set described as the intersection of finitely many half-spaces, i.e., a polyhedron. Formally, we define an instance of a linear program as follows.

1. Without loss of generality, we choose minimization instead of maximization.

Definition 2 (Linear programming instance) *An instance of a* linear program *(LP) is a tuple (A, b, c), where A is a matrix in $\mathbb{Q}^{m\times n}$, and b and c are vectors in $\mathbb{Q}^m$ and $\mathbb{Q}^n$, respectively.*

The associated optimization problem asks to minimize a linear functional or linear objective over a polyhedron.[2] That is, we aim at finding a vector x in $\mathbb{Q}^n$ that minimizes $c^T x$ over the *feasible set*

$$X = \{x \in \mathbb{Q}^n \mid A_j x \leq b_j \text{ for } j \in [m] \text{ and } x_i \geq 0 \text{ for } i \in [n]\}.$$

In practice, LPs are solved using the Simplex method or polynomial-time interior-point methods (Bertsimas and Tsitsiklis, 1997). Due to their continuous nature, LPs cannot encode the feasible set of a CO problem. Hence, we extend LPs by adding *integrality constraints*, requiring that the value assigned to each variable is an integer. Consequently, we aim to find the vector x in $\mathbb{Z}^n$ that minimizes $c^T x$ over the feasible set

$$X = \{x \in \mathbb{Z}^n \mid A_j x \leq b_j \text{ for } j \in [m], x_i \geq 0 \text{ and } x_i \in \mathbb{Z} \text{ for } i \in [n]\}$$

Such integer linear optimization problems are solved by tree search algorithms, e.g., branch and cut algorithms, see Section 3.2 for details. By dropping the integrality constraints, we again obtain an instance of an LP, which we call *relaxation*.

Example 2 *We provide an ILP that encodes the optimal solution to the TSP. Essentially, it encodes the order of the nodes or cities within its variables. Thereto, let*

$$x_{ij} = \begin{cases} 1 & \text{if the cycle goes from city } i \text{ to city } j, \\ 0 & \text{otherwise.} \end{cases}$$

and let $w_{ij} > 0$ be the cost or distance of travelling from city i to city j, $i \neq j$. Then the TSP can be written as the following ILP:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} \sum_{j\neq i, j=1}^{n} w_{ij} x_{ij} & \\ \text{subject to} \quad & \sum_{i=1, i\neq j}^{n} x_{ij} = 1 & j \in [n], \\ & \sum_{j=1, j\neq i}^{n} x_{ij} = 1 & i \in [n], \\ & \sum_{i\in Q} \sum_{j\neq i, j\in Q} x_{ij} \geq 1 & \forall Q \subsetneq [n], |Q| \geq 2. \end{aligned}$$

The first two constraints encode that each city should have exactly one in-going and out-going edge, respectively. The last constraint makes sure that all cities are within the same tour, i.e., there exist no sub-tours, and the returned solution is not a union of smaller tours.

2. In the above definition, we assumed that the LP is feasible, i.e., $X \neq \emptyset$, and that a finite minimum value exists. In what follows, we assume that both conditions are always fulfilled.

In practice, one often faces problems consisting of a mix of integer and continuous variables. These are commonly known as *mixed-integer programs* (MIPs). Formally, given an integer $p > 0$, MIPs aim at finding a vector x in $\mathbb{Q}^n$ that minimizes $c^T x$ over the *feasible set*

$$X = \{x \in \mathbb{Q}^n \mid A_j x \leq b_j \text{ for } j \in [m], x_i \geq 0 \text{ for } i \in [n], \text{ and } x \in \mathbb{Z}^p \times \mathbb{Q}^{n-p}\}.$$

Here, n is the number of variables we are optimizing, out of which p are required to be integers.

2.3.2 SAT

The *Boolean satisfiability problem* (SAT) asks, given a boolean formula or propositional logic formula, if there exists a variable assignment (assign *true* or *false* to variables) such that the formula evaluates to *true*. Hence, formally we can define it as follows.

Definition 3 (SAT)
Input: n propositional logic formula φ with variable set V.
Output: Yes*, if there exists a variable assignment $A\colon V \to \{true, false\}$ such that the formula φ evaluates to* true*;* No *otherwise.*

The SAT was the first problem to be shown to be NP-complete, however modern solvers routinely solve industrial-scale instances in practice (Prasad et al., 2005).

Example 3
Input: The propositional logic formula $\varphi = x_1 \wedge \neg x_2 \vee x_2 \wedge \neg x_3$.
Output: Yes*, since setting $A(x_1) = false$, $A(x_2) = true$, and $A(x_3) = false$ evaluates to* true*.*

2.3.3 Constraint satisfaction and optimization problems

This section presents both *constraint satisfaction problems* and *constraint optimization problems*, the most generic way to formalize CO problems. Formally, an instance of a *constraint satisfaction problem* is defined as follows.

Definition 4 (Constraint satisfaction problem instance) *An instance of a* constraint satisfaction problem *(CSP) is a tuple $\big(X, D(X), C\big)$, where X is the set of variables, $D(X)$ is the set of domains of the variables, and C is the set of constraints that restrict assignments of values to variables. A solution is an assignment of values from D to X that satisfies all the constraints of C.*

A natural extension of CSPs are *constrained optimization problems*, that also have an objective function. The goal becomes that of finding a feasible assignment that minimizes the objective function. The main difference with the previous optimization frameworks is that constrained optimization problems do not require underlying assumptions on the variables, constraints, and objective functions. Unlike MIPs, non-linear objectives and constraints are applicable within this framework. For instance, a TSP model is presented next.

Example 4 *Given a configuration with n cities and a weight matrix w in $\mathbb{Q}^{n \times n}$, the TSP can be modeled using n variables x_i over the domains $D(x_i)\colon [n]$. Variable x_i indicates the i-th city*

to be visited. The objective function and constraints are as follows, where allDifferent(X) *enforces that each variable from X takes a different value (Régin, 1994). Note also that the entries of the weight matrix w are indexed using variables:*

$$\min\ w_{x_n,x_1} + \sum_{i=1}^{n-1} w_{x_i,x_{i+1}}$$
$$\text{subject to}\ \ \text{allDifferent}(x_1,\ldots,x_n).$$

This model enforces each city to have another city as a successor and sums up the distances between each pair of consecutive cities along the cycle.

As shown in the above example, constrained problems can model arbitrary constraints and objective function. This generality makes it possible to use general-purpose solving methods such as *local search* or *constraint programming*.

Constraint programming (Rossi et al., 2006, CP) CP is a general framework proposing simple algorithmic solutions to constrained problems. It is a complete approach, meaning it is possible to prove the optimality of the solutions found. The solving process consists of a complete enumeration of all possible variable assignments until the best solution has been found. To cope with the implied (exponentially) large search trees, one utilizes a mechanism called *propagation*, which reduces the number of possibilities. Here, the propagation of the constraint c removes values from domains violated by c. This process is repeated at each domain change and for each constraint until no value exists anymore. The efficiency of a CP solver relies heavily on the quality of their propagators. The CP search commonly proceeds in a depth-first fashion, together with branch and bound. For each found feasible solution, the solver adds a constraint, ensuring that the following solution has to be better than the current one. Upon finding an infeasible solution, the search backtracks to the previous decision. With this procedure, and provided that the whole search space has been explored, the final solution found is then guaranteed to be optimal.

Local search (Aarts and Lenstra, 2003) Local search is another algorithmic framework that is commonly used to solve general, large-scale constrained problems. Unlike CP, local search only partially explores the solution space in a perturbative fashion, and is thus an incomplete approach that does not provide an optimality guarantee on the solution it returns. In its simplest form, the search starts from a candidate solution s and iteratively explores the solution space by selecting a neighboring solution until no improvement occurs. Here, the *neighborhood* of a solution is the set of solutions obtained by making some modifications to the solution s. In practice, local search algorithms are improved with metaheuristics (Glover and Kochenberger, 2006), such as *simulated annealing* (Van Laarhoven and Aarts, 1987), *tabu search* (Glover and Laguna, 1998) or *variable neighborhood search* (Mladenović and Hansen, 1997), all of which are designed to help escape *local minima*.

2.4 Machine Learning

Here we give a short and concise overview of machine learning. We cover the three main branches of the field, i.e., *supervised learning*, *unsupervised learning*, and *reinforcement learning*. For details, see (Mohri et al., 2012; Shalev-Shwartz and Ben-David, 2014). Moreover, we introduce *imitation learning*, which is of high relevance to CO.

Supervised learning Given a finite training set, i.e., a set of examples (e.g., graphs) together with target values (e.g., real values in the case of regression), supervised learning tries to adapt the parameters of a model (e.g., a neural network) based on the examples and targets. The adaptation of the parameters is achieved by minimizing a loss function, which measures how well the chosen parameters align with the target values. Formally, let $\mathcal{X}$ be the set of possible *examples* and let $\mathcal{Y}$ be the set of possible *target values*. We assume that the pairs in $\mathcal{X} \times \mathcal{Y}$ are independently and identically distributed with respect to a fixed but unknown distribution $\mathcal{D}$. Moreover, we assume that there exists a *target concept* $c\colon \mathcal{X} \to \mathcal{Y}$ which maps each example to its target value. Given a sample $S = ((s_1, c(s_1)), \ldots, (s_m, c(s_m)))$ drawn i.i.d. from $\mathcal{D}$, the aim of supervised machine learning is to select a *hypothesis* $h\colon \mathcal{X} \to \mathcal{Y}$ from the set of possible hypotheses by minimizing the *empirical error* $\widehat{R}(h) = \frac{1}{m}\sum_{i=1}^{m} l(h(s_i), c(s_i))$, where $l\colon \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is the loss function. To avoid overfitting to the given samples, we add a *regularization penalty* $\Omega\colon H \to \mathbb{R}$ to the empirical error. Examples of supervised machine learning methods include neural networks, support vector machines, and boosting.

Unsupervised learning Unlike supervised learning, there is no training set in the unsupervised case, i.e., no target values are available. Accordingly, unsupervised learning aims to capture representative characteristics of the data (features) by minimizing an unsupervised loss function, $l\colon \mathcal{X} \to \mathbb{R}$. In this case, the loss function only directly depends on the input samples s_i, as no labels are provided upfront. Examples of unsupervised machine learning methods include autoencoders, clustering, and principal component analysis.

Reinforcement learning (RL) Similarly to unsupervised learning, reinforcement learning does not rely on a labeled training set. Instead, an *agent* explores an environment, e.g., a graph, by taking *actions*. To guide the agent in its exploration, it receives two types of feedback, its current *state*, and a *reward*, usually a real-valued scalar, indicating how well it achieved its goal so far. The RL agent aims to maximize the cumulative reward it receives by determining the best actions. Formally, let (S, A, T, R) be a tuple representing a *Markov decision process* (MDP). Here S is the set of *states* in the environment and A is the set of *actions* that the agent can do. The function $T\colon S \times S \times A \to [0, 1]$ is the *transition probability function* giving the probability, $T(s, s', a)$, of transitioning from s to s' if action a is performed, such that $\sum_{s'}^{S} T(s, s', a) = 1$ for all s in S and a in A. Finally, $R\colon S \times A \to \mathbb{R}$ is the *reward function* of taking an action from a specific state. An agent's behavior is defined by a *policy* $\pi\colon S \times A \to [0, 1]$, describing the probability of taking an action from a given state. From an initial state s_1, the agents perform actions, yielding a sequence of states $\{s_t\}_t$ until reaching a terminal state, s_Θ. Such a sequence $s_1 \ldots, s_\Theta$ is referred to as an *episode*. An agent's goal is to learn a policy maximizing the cumulative sum of rewards, eventually discounted by a value γ in $[0, 1]$, during an episode, i.e., $\sum_{k=1}^{\Theta} \gamma^k R(s_k, a_k)$ is maximized. While such a learning setting is very general, the number of combinations increases exponentially with the number of states and actions, quickly making the problem intractable. Excluding hybrid approaches, e.g., RL with Monte Carlo tree search (Browne et al., 2012) and model-based approaches (Polydoros and Nalpantidis, 2017), there exist two kinds of reinforcement learning algorithms, *value-based methods*, aiming to learn a function characterizing the goodness of each action, and *policy-based methods*, aiming to learn the policy directly.

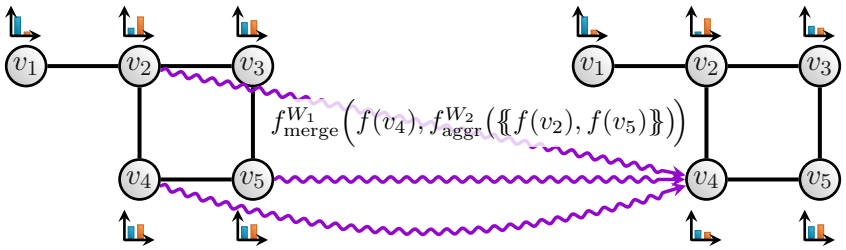


Figure 3: Illustration of the neighborhood aggregation step of a GNN around node v_4.

Imitation learning Imitation learning (Ross, 2013) attempts to solve sequential decision-making problems by imitating another ("expert") policy rather than relying on rewards for feedback as done in RL. This makes imitation learning attractive for CO because, for many control problems, one can devise rules that make excellent decisions but are not practical because of computational cost or because they cheat by using information that would not be available at solving time.

Imitation learning algorithms can be offline or online. When offline, examples of expert behavior are collected beforehand, and the student policy's training is done subsequently. In this scenario, training is simply a form of supervised learning. When online, however, the training occurs while interacting with the environment, usually by querying the expert for advice when encountering new states. Online algorithms can be further subdivided into on-policy and off-policy algorithms. In on-policy algorithms, the distribution of states from which examples of expert actions were collected matches the student policy's stationary distribution to be updated. In off-policy algorithms, there is a mismatch between the distribution of states from which the expert was queried and the distribution of states the student policy is likely to encounter. Some off-policy algorithms attempt to correct this mismatch accordingly.

2.5 Graph Neural Networks

Intuitively, GNNs compute a vectorial representation, i.e., a d-dimensional real vector, representing each node in a graph by aggregating information from neighboring nodes; see Figure 3 for an illustration. Formally, let (G, l) be a labeled graph with an initial node coloring $f^{(0)} \colon V(G) \to \mathbb{R}^{1\times d}$ that is *consistent* with l. This means that each node v is annotated with a feature $f^{(0)}(v)$ in $\mathbb{R}^{1\times d}$ such that $f^{(0)}(u) = f^{(0)}(v)$ if $l(u) = l(v)$. Alternatively, $f^{(0)}(v)$ can be an arbitrary real-valued feature vector associated with v, such as a cost function of a CO problem. A GNN model consists of a stack of neural network layers. Each layer aggregates local neighborhood information, i.e., neighbors' features, within each node and then passes this aggregated information to the next layer.

GNNs are often realised as follows (Morris et al., 2019): in each layer $t > 0$, we compute new features

$$f^{(t)}(v) = \sigma\Big(f^{(t-1)}(v) \cdot W_1^{(t)} + \sum_{w\in N(v)} f^{(t-1)}(w) \cdot W_2^{(t)}\Big) \tag{1}$$

in $\mathbb{R}^{1\times e}$ for v, where $W_1^{(t)}$ and $W_2^{(t)}$ are parameter matrices from $\mathbb{R}^{d\times e}$, and σ denotes a component-wise non-linear function, e.g., a sigmoid or a ReLU.[3]

3. For clarity of presentation, we omit biases.

Following Gilmer et al. (2017), one may also replace the sum defined over the neighborhood in the above equation by a permutation-invariant, differentiable function. One may substitute the outer sum, e.g., by a column-wise vector concatenation. Thus, in full generality, a new feature $f^{(t)}(v)$ is computed as

$$f_{\text{merge}}^{W_1}\Big(f^{(t-1)}(v), f_{\text{aggr}}^{W_2}\big(\{\!\{f^{(t-1)}(w) \mid w \in N(v)\}\!\}\big)\Big), \tag{2}$$

where $f_{\text{aggr}}^{W_1}$ aggregates over the multiset of neighborhood features and $f_{\text{merge}}^{W_2}$ merges the node's representations from step $(t-1)$ with the computed neighborhood features. Both $f_{\text{aggr}}^{W_1}$ and $f_{\text{merge}}^{W_2}$ may be arbitrary, differentiable, permutation-invariant functions and, by analogy to Equation 1, we denote their parameters as W_1 and W_2, respectively. To adapt the parameters W_1 and W_2 of Equations 1 and 2, they are optimized in an end-to-end fashion (usually via stochastic gradient descent) together with the parameters of a neural network used for classification or regression.

3. GNNs for Combinatorial Optimization: The State of the Art

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. It is not uncommon to combine the GNN's variable or constraint representations with hand-crafted features, which would otherwise be challenging to extract automatically with the GNN. Coupled with an appropriate ML paradigm (Section 2.4), GNNs have been shown to guide exact and heuristic algorithms towards finding good feasible solutions faster (Section 3.1). GNNs have also been used to guide certifying optimality/infeasibility more efficiently (Section 3.2). In this case, GNNs are usually integrated with an existing complete algorithm, because an optimality certificate has in general exponential size concerning the problem description size, and it is not clear how to devise GNNs with such large outputs. 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, enabling improved generalization and interpretability, as we will discuss in Section 3.3.

3.1 On the Primal Side: Finding Feasible Solutions

We begin by discussing the use of GNNs in improving the solution-finding process in CO. It is natural to wonder whether the primal side of CO is worth exploring when, given sufficient time, exact (or complete) algorithms guarantee finding an optimal solution. The following practical scenarios motivate the need for quickly obtaining high-quality feasible solutions.

a) Optimality guarantees are often not needed A practitioner may only be interested in the quality of a feasible solution in absolute terms rather than relative to the (typically unknown) optimal value of a problem instance. To assess a heuristic's suitability in this scenario, one can evaluate it on a set of instances for which the optimal value is known. However, when used on a new problem instance, the heuristic's solution can only be

assessed via its (absolute) objective value. This situation arises when the CO problem of interest is practically intractable with an exact solver. For example, many vehicle routing problems admit strong MIP formulations that have an exponential number of variables or constraints, similar to the TSP formulation in Example 2, see (Toth and Vigo, 2014). While such problems may be solved exactly using column or constraint generation (Dror et al., 1994), a heuristic that consistently finds good solutions within a short user-defined time limit may be preferable.

b) Optimality is desired, but quickly finding a good solution is the priority Because optimality is still of interest here, one would like to use an exact solver that is focused on the primal side. A common use case is to take a good solution and start analyzing it manually in the current application context while the exact solver keeps running in the background. An early feasible solution allows for fast decision-making, early termination of the solver, or even revisiting the mathematical model with additional constraints that were initially ignored. MIP solvers usually provide a parameter that can be set to emphasize finding solutions quickly; see CPLEX's emphasis switch parameter for an example.[4] Among other measures, these parameters increase the time or iterations allotted to primal heuristics at nodes of the search tree, which improves the odds of finding a good solution early on in the search.

Alternatively, one could also develop a custom, standalone heuristic that is executed first, providing a warm start solution to the exact solver. This simple approach is widely used and addresses both goals a) and b) simultaneously when the heuristic in question is effective for the problem of interest. This can also be done in order to obtain a high-quality first solution for initiating a local search.

Next, we will discuss various approaches that leverage GNNs in the primal setting. We will also touch on some approaches that have leveraged machine learning without a GNN but could benefit from the latter.

3.1.1 Learning heuristics from scratch

The works in this section attempt to construct a feasible solution "from scratch", in that they do not use a constraint, linear or integer programming solver to help the machine learning model satisfy the problem's constraints. The approaches herein either deal with combinatorial problems with simple constraints or provide a mechanism that guarantees that the output solution is feasible. The TSP, see Example 1, has received substantial attention from the machine learning community following the work of Vinyals et al. (2015). The authors use a sequence-to-sequence "pointer network" (Ptr-Net) to map two-dimensional points to a tour of small total length. The Ptr-Net was trained with supervised learning and thus required near-optimal solutions as labels; this may be a limiting factor when the TSP instances of interest are hard to solve and thus to label. To overcome the need for near-optimal solutions, Bello et al. (2017) use reinforcement learning to train Ptr-Net models for the TSP. Although the use of RL in combinatorial problems had been explored much earlier, e.g., by Zhang and Dietterich (1995), the work of Bello et al. (2017) was one of

4. `https://www.ibm.com/support/knowledgecenter/SSSA5P_20.1.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/MIPEmphasis.html`

the first to combine RL with deep neural networks in the CO setting. However, it failed to address a fundamental limitation of Ptr-Nets. A Ptr-Net deals with sequences as its inputs and outputs, whereas a solution to the TSP has no natural ordering and is better viewed as a set of edges that form a valid tour.

In (Khalil et al., 2017), GNNs were leveraged for the first time in the context of graph optimization problems, addressing this last limitation. The GNN served as the function approximator for the value function in a Deep Q-learning (DQN) formulation of CO on graphs. The authors use a Structure2Vec GNN architecture (Dai et al., 2016), similar to Equation (1), to embed nodes of the input graph. Through the combination of GNN and DQN, a greedy node selection policy—S2V-DQN—is learned on a set of problem instances drawn from the same distribution. In this context, the TSP can be modeled as a graph problem by considering the weighted complete graph on the cities, where the edge weights are the distances between a pair of cities. A greedy node selection heuristic for the TSP iteratively selects nodes, adding the edge connecting every two consecutively selected nodes to the final tour. As such, a feasible solution is guaranteed to be obtained after $n-1$ greedy node selection steps, where the first node is chosen arbitrarily, and n is the number of nodes or cities of a TSP instance. Because embedding the complete graph with a GNN can be computationally expensive and possibly unnecessary to select a suitable node, a k-nearest neighbor graph can be used instead of the complete graph. Khalil et al. (2017) apply the above approach to other classical graph optimization problems such as Maximum Cut (Max-Cut) and Minimum Vertex Cover (MVC).

Additionally, they extend the framework to the Set Covering Problem (SCP), in which a minimal number of sets must be selected to cover a universe of elements. While the SCP is not typically modeled as a graph problem it can be naturally modeled as a bipartite graph, see Khalil et al. (2017), enabling the use of GNNs as in TSP, MVC, and Max-Cut. More broadly, the reducibility among NP-complete problems (Karp, 1972) guarantees that a polynomial-time transformation between any two NP-complete problems exists. Whether such a transformation is practically tractable (e.g., a quadratic or cubic-time transformation might be considered too expensive) or whether the greedy node selection approach makes sense will depend on the particular combinatorial problem at hand. However, the approach introduced in (Khalil et al., 2017) seems to be useful for a variety of problems and admits many direct extensions and improvements, some of which we will survey next.

We categorize contributions in this space along three axes. The CO problems being addressed, the use of custom GNN architectures that improve over standard ones in some respect, and specialized training approaches that alleviate bottlenecks in typical supervised or reinforcement learning for CO. Cross-cutting contributions will be highlighted.

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), a well-known GNN architecture, using an Actor-Critic RL approach. Problems tackled in (Kool et al., 2019) include the TSP, capacitated VRP (CVRP), the Orienteering Problem (OP), and the Prize-Collecting TSP (PCTSP). Nazari et al. (2018) also tackle the CVRP with a somewhat similar encoder-decoder approach. Prates et al. (2019) train a GNN in a supervised manner to predict the satisfiability of the decision version of the TSP. Instances up to 105 cities are considered.

The problems discussed thus far in this section have constraints that are relatively easy to satisfy. For example, a feasible solution to a TSP instance is simply a tour on all nodes, implying that a constructive policy should only consider nodes, not in the current partial solution, and terminate as soon as a tour has been constructed. These requirements can be enforced by restricting the RL *action space* appropriately. As such, the training procedure and the GNN model need to focus exclusively on optimizing the average objective function of the combinatorial problem while enforcing these "easy" constraints by manually constraining the action space of the RL agent. In many practical problems, some of the constraints may be trickier to satisfy. Consider the more general TSP with Time Windows (Savelsbergh, 1985, TSPTW), 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. It is easy to see how a constructive policy may "get stuck" in a state or partial solution in which all actions are infeasible. Ma et al. (2019) tackle the TSPTW by augmenting the building blocks we have discussed so far (GNN with RL) with a hierarchical perspective. Some of the learnable parameters are responsible for generating feasible solutions, while others focus on minimizing 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. Also using RL, Cappart et al. (2020) take another direction and propose to tackle problems that are hard to satisfy (such as the TSPTW) by reward shaping. The reward signal they introduce has two specific and hierarchic goals: firstly, finding a feasible and complete solution, and secondly, find the solution minimizing the objective function among the feasible solutions. The construction of a solution is stopped as soon as there is no action available, which correspond to an infeasible partial solution. Each complete solution obtained has then the guarantee to be feasible.

Liu et al. (2019) 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 problems in an end-to-end fashion. 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. (2018); Kool et al. (2019); Joshi et al. (2019); Khalil et al. (2017) can be efficiently used as the first solution of a local search procedure for solving the TSP.

Fey et al. (2020); Li et al. (2019) 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 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.

3.1.2 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. Li et al. (2018) 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.

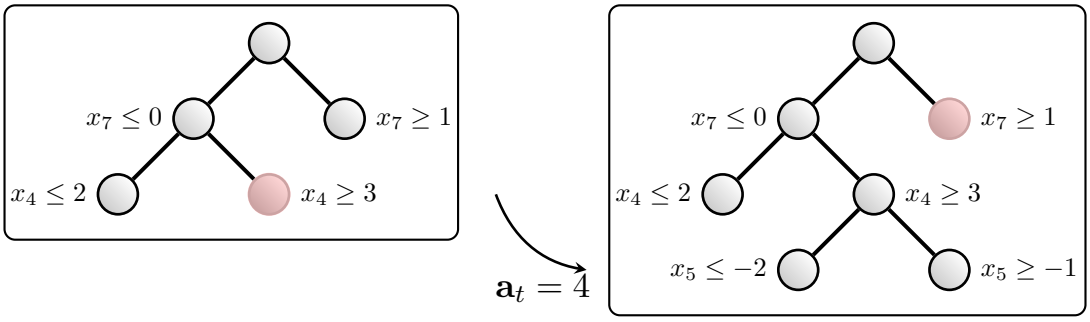


Figure 4: Variable selection in the branch and bound integer programming algorithm as a MDP.

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, constraint, 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 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. They propose to use REINFORCE parameterized by a GNN on the above graph to learn a valid assignment on a variety of problems, e.g., 3-SAT, clique detection, and graph coloring. To combat the sparse reward nature of SAT problems, they additionally employ curriculum learning (Bengio et al., 2009).

3.2 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. This problem can be assimilated to the task of finding the optimal policy of a MDP, as illustrated in Figure 4. 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 3.1) 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.

A similar branch and bound algorithm can be employed in neural network verification, where properties of a neural network are verified by solving a mixed-integer optimization problem. Lu and Kumar (2020) propose to represent the neural network to be verified as a graph with attributes and train a GNN to imitate strong branching. The approach is thus close to the one of Gasse et al. (2019), although the graphs and the GNN architecture are specifically designed for neural network verification, showing state-of-the-art improvements over hand-designed methods.

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) 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) propose to model quantified boolean formulas as bipartite graphs and teach a GNN to branch using REINFORCE. Although the reinforcement learning algorithm used is very simple, they achieve substantial improvements in number of formulas solved within a given time limit compared to VSIDS, the standard

branching heuristic. Two other works applied similar ideas to related problems. Kurin et al. (2020) model propositional Boolean formulas as bipartite graphs and train a GNN to branch with Q-learning. Although the problem is different, they similarly show that a the learned heuristic can improve on VSIDS, namely in the number of iterations needed to solve a given problem. Moreover, Vaezipoor et al. (2020) represent propositional Boolean formulas as bipartite graphs and train a GNN to branch with evolution strategies, but in a Davis–Putnam–Logemann–Loveland solver for #SAT, the counting analog of SAT. They show this yields improvements in solving time compared to SharpSAT, a state-of-the-art exact method. Finally, in a different direction, Selsam and Bjørner (2019) propose to 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 propose to 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, such as branch and bound, iterative limited discrepancy search, and restart-based search that work by 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 MDPs on graph states, such as the TSP with time windows, Cappart et al. (2020) propose to train a GNN to learn a good policy or action-value function for the Markov decision process using reinforcement learning. The resulting model is used 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. To do so, they propose to represent a CSP as a simple and undirected tripartite graph, on which each variable, possible value, and constraint is represented by a node. The nodes are connected by an edge if and only if a variable is involved in a constraint, or if a value is inside the domain of a variable.

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 (corresponding to the shortest path in the graph) 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) propose to 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. The GNN architecture used and the problem representation as a graph is similar as the one proposed by Khalil et al. (2017).

3.3 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. For example, merely increasing the test input size, e.g., the number of nodes in the input graph, is often sufficient to lose most of the training's predictive power.

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 ones within the training distribution. Therefore, unless we can accurately foreshadow the kinds of inputs our neural CO approach will be solving, it could be essential to address the issue of out-of-distribution generalization in neural networks meaningfully.

It is also important to clarify what we mean by extrapolation in this sense. Many CO problems of interest are NP-hard and, therefore, likely to be out of reach of GNN computation. This is because decision problems solvable by end-to-end GNNs of tractable depth (i.e., polynomial in the number of nodes) are necessarily in P, by definition. Hence, when we say a GNN extrapolates on a hard CO task, we will not imply that the GNN will produce correct solutions for arbitrarily large inputs—unless P=NP, this is a hopeless ask. Instead, we will require the GNN to produce solutions that *align* with an appropriate polynomial-time heuristic. For example, using the techniques we will present in this section, it is feasible to train GNNs that will produce 2-OPT approximations for TSP (Croes, 1958).

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 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.

Prior art in the area has investigated the construction of general-purpose neural computers, e.g., the neural Turing machine (Graves et al., 2014) and the differentiable neural computer (Graves et al., 2016). While such architectures have all the hallmarks of general computation, they introduced several components at once, making them often challenging to optimize, and in practice, they are almost always outperformed by simple relational reasoners (Santoro et al., 2017, 2018). More carefully constructed variants and inductive biases for learning to execute (Zaremba and Sutskever, 2014) have also been constructed, but mainly focusing on primitive algorithms (such as arithmetic). Prominent examples include the neural GPU (Kaiser and Sutskever, 2015), neural RAM (Kurach et al., 2015), neural programmer-interpreters (Reed and De Freitas, 2015), and neural arithmetic-logic units (Trask et al., 2018; Madsen and Johansen, 2020).

Powered by the rapid development of GNNs, algorithmic reasoning experienced a strong resurgence, tackling combinatorial algorithms of superlinear complexity with graph-structured processing at the core. Initial theoretical analyses (Xu et al., 2019b) demonstrate why this is

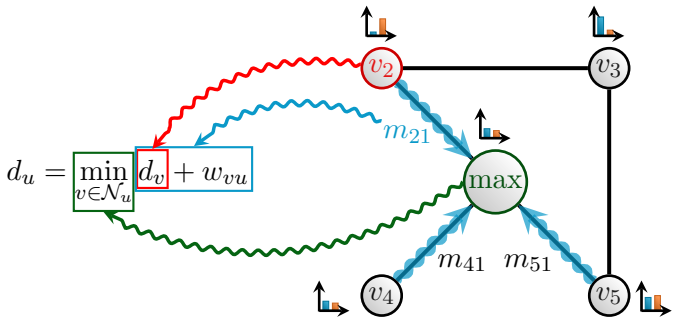


Figure 5: Illustration of algorithmic alignment, in the case of the Bellman-Ford shortest path-finding algorithm (Bellman, 1958). It computes distance estimates for every node, d_u, and is shown on the left. Specifically, a GNN aligns well with this dynamic programming update. Node features align with intermediate computed values (**red**), message functions align with the candidate solutions from each neighbor (**blue**), and the aggregation function (if, e.g., chosen to be max) aligns with the optimization across neighbours (**green**). Hence, GNNs with max aggregation are likely appropriate for problems that require forms of path-finding.

a good idea, GNNs align with dynamic programming (Bellman, 1966), which is a language in which most algorithms can be expressed. Hence, it is viable that most polynomial-time combinatorial reasoners of interest will be modelable using a GNN. We will now investigate alignment in more detail.

3.3.1 Algorithmic alignment

The concept of *algorithmic alignment* introduced by 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. Throughout this section, we will use Figure 5 as a guiding example.

Guided by this principle, novel GNN architectures and training regimes have been recently proposed to facilitate aligning with broader classes of combinatorial algorithms. As such, those works concretize the theoretical findings of Xu et al. (2019b).

The work of Veličković et al. (2020b) on the 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.

Using the *encode-process-decode* paradigm (Hamrick et al., 2018) Inputs, x in $\mathcal{X}$, are encoded into latents, z in $\mathcal{Z}$, using an encoder (G)NN, $f\colon \mathcal{X} \to \mathcal{Z}$. Latents are decoded into outputs, y in $\mathcal{Y}$, using a decoder (G)NN, $g\colon \mathcal{Z} \to \mathcal{Y}$, and computation in the latent space is performed by a processor GNN, $P\colon \mathcal{Z} \to \mathcal{Z}$, which is typically executed over a certain (fixed or inferred) number of steps. This aligns well with iterative computation commonly found in most classical algorithms. Further, we may

observe the processor network as a latent-space algorithm, which proves very useful for both *algorithmic reuse* and *multi-task learning*.

Favoring the *max* aggregation function This aligns well with the fact most combinatorial algorithms require some form of local decision-making, e.g., "which neighbor is the predecessor along the shortest path?". Moreover, max aggregation is generally more stable at larger scales (as the effective node degree of a max-aggregated GNN is $O(d)$ for d-dimensional feature representations). Such findings have been independently verified (Joshi et al., 2020; Richter and Wattenhofer, 2020; Corso et al., 2020) and contradict the more common advice to use the sum-aggregator (Xu et al., 2019a).

Leveraging *strong supervision* with *teacher forcing* (Williams and Zipser, 1989) If, at training time, we have access to rollouts (or "hints") from the ground truth algorithm, which illustrates how input data is manipulated[5] throughout that algorithm's execution, these can be used as auxiliary supervision signals, and further, the model may be asked only to predict one-step manipulations. Such an imitation learning setting can substantially improve out-of-distribution performance, as the additional supervision acts as a strong regularizer, constraining the function learned by the processor to more closely follow the ground-truth algorithm's iterations. This provides a mechanism for encoding and aligning with invariants, e.g., after k iterations of a shortest-path algorithm such as Bellman-Ford (Bellman, 1958), it should be possible to compute shortest paths that use up to k hops from the source node. Strong supervision works well even without hints, as was demonstrated by the RRN model of Palm et al. (2017). Therein, the authors achieve "convergent message passing" by supervising a GNN to decode the ground-truth output at every step of execution.

***Masking* of outputs (and, by extension, loss functions)** GNNs are capable of processing all objects in a graph simultaneously—but for many combinatorial reasoning procedures of interest, this is overkill. Many efficient combinatorial algorithms are efficient precisely because they focus on only a small amount of nodes (typically on the order of $O(\log n)$) at each iteration, leaving the rest unchanged. Explicitly making the neural network predict which nodes are relevant to the current step (via a learnable mask) can therefore be strongly impactful, and at times more important than the choice of processor.[6]

Executing multiple related algorithms In this case, the processor network is shared across algorithms and becomes a multi-task learner, either simultaneously or in a curriculum (Bengio et al., 2009). When done properly, this can positively reinforce the pair-wise relations between algorithms, allowing for combining multiple heuristics into one reasoner (as done by Vrček et al. (2020)) or using the output of simpler algorithms as "latent input" for more complex ones, significantly improving empirical performance on the complex task.

5. In this sense, for sequences of unique inputs all correct sorting algorithms have the same input-output pairs, but potentially different sequences of hints.
6. For example, Veličković et al. (2020b) has shown that, for learning minimum spanning tree algorithms, LSTM processors with the masking inductive bias performed significantly better out-of-distribution than GNN processors without it.

While initially applied to simple path-finding and spanning-tree algorithms, the prescriptions listed above have made their way into bipartite matching (Georgiev and Lió, 2020), min-cut problems (Awasthi et al., 2021), model-based planning (Deac et al., 2020a) and heuristics for Hamiltonian paths (Vrček et al., 2020), and the TSP (Joshi et al., 2020).

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 in the findings 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. PrediNets (Shanahan et al., 2020) align with computations of propositional logic. IterGNNs (Tang et al., 2020) provably align well with iterative algorithms, adaptively learning a stopping criterion without requiring an explicit termination network. HomoGNNs (Tang et al., 2020) remove all biases from the GNN computation, making them align well with *homogeneous functions*. These are functions exhibiting multiplicative scaling behavior—i.e. for any $\lambda \in \mathbb{R}$, $f(\lambda x) = \lambda f(x)$—a property held by many combinatorial tasks.[7] Neural shuffle-exchange networks (Freivalds et al., 2019; Draguns et al., 2020) directly fix connectivity patterns between nodes based on results from routing theory (such as Beneš networks), 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.

As a motivating example, PGNs tackle *incremental connectivity* (Figure 6). Answering queries on whether pairs of nodes are connected under the constraint that edges may only ever be added into the graph. It is easy to construct a worst-case "path graph" for which query answering would require $O(n)$ GNN steps. PGNs instead learn to imitate edges of a disjoint-set union (DSU) data structure (Galler and Fisher, 1964). DSUs efficiently represent sets of connected components, allowing for connectivity querying in $O(\alpha(n))$ amortised complexity (Tarjan, 1975), where α is the inverse Ackermann function—essentially, a constant for all astronomically sensible values of n. Thus, by carefully choosing auxiliary edges for the processor GNN, PGNs can significantly improve on the prior art in neural execution. They algorithmically align with any iterative algorithmic computation backed by a data structure.

All of the executors listed above focus on performing message passing over *exactly* the nodes provided by the input graph, never modifying this node set during execution. This fundamentally limits them to simulating algorithms with up to $O(1)$ auxiliary space per node. The persistent message passing (PMP) model of Strathmann et al. (2021) has lifted this restriction: by taking inspiration from persistent data structures (Driscoll et al., 1989), PMP allows the GNN to selectively *persist* their nodes' state after every step of message passing. Now the nodes' latent state is never overwritten; instead, a copy of the persisted

7. For example, if all the edge weights in a shortest path problem are multiplied by λ, *any* path length—including the shortest path length—also gets multiplied by λ.

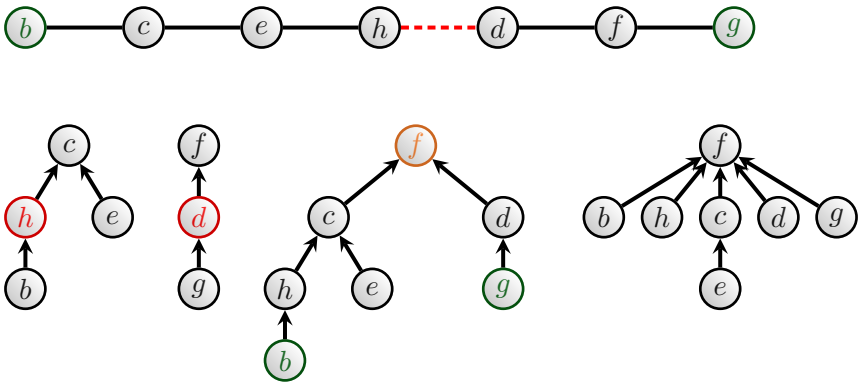


Figure 6: The utility of dynamically choosing the graph to reason over for incremental connectivity. It is easy to construct an example *path graph* (*top*), wherein deciding connectivity can require linearly many GNN iterations. This can be ameliorated by reasoning over different links—namely, ones of the disjoint set union (DSU) data structure (Galler and Fisher, 1964) that represents each connected component as a *rooted tree*. At the bottom, from left-to-right, we illustrate the evolution of the DSU for the graph above, once the edge (h, d) is added and query (b, g) executed. Note how the DSU gets compressed after each query (Tarjan, 1975), thus making it far easier for subsequent querying of whether two nodes share the same root.

nodes is performed, storing their new latents. This effectively endows PMP with an episodic memory (Pritzel et al., 2017) of its past computations, and has the potential to overcome more general problematic aspects in learning GNNs, such as oversmoothing.

3.3.2 Perspectives and outlooks

While neural algorithmic reasoning has resurged only recently, it has already covered much ground, building GNN-style architectures that align well with dynamic programming, iterative computation[8], as well as algorithms backed by data structures. This already is able to support many essential constructs from theoretical computer science; given that such primitives are now introduced only gradually rather than all at once (with each paper carefully studying one form of algorithmic alignment), we are getting closer to re-imagining the differentiable neural computer with substantially more stable components.

Further, recent theoretical results have provided a unifying explanation for why algorithmically inspired prescriptions provide benefits to extrapolating both in algorithmic and in physics-based tasks (Xu et al., 2020). Specifically, the authors make a useful geometric argument—ReLU-backed MLPs always tend to extrapolate linearly outside of the support of the training set. Hence, if we can design architecture components or task featurisations such that the individual parts (e.g., message functions in GNNs) have to learn roughly-linear ground-truth functions, this theoretically and practically implies stronger out-of-distribution performance. This explains, e.g., why *max* aggregation performs well for shortest path-finding.

8. Recent work (Yang et al., 2021) has also demonstrated that GNNs can be made to align with *iterative optimisation* algorithms, such as proximal gradient descent and iterative reweighted least squares.

The Bellman-Ford dynamic programming rule (e.g., as in Figure 5)

$$d_u = \min_{v \in \mathcal{N}_u} d_v + w_{vu} \tag{3}$$

is an edge-wise linear function followed by a minimisation. Hence, assuming a GNN of the form

$$h'_u = \max_{v \in \mathcal{N}_u} M(h_u, h_v, w_{vu}), \tag{4}$$

we can see that the message function M now has to learn a *linear function* in h_v and w_{vu}—a substantially easier feat than if the sum-aggregation is used.

While all of the above dealt with improving the performance of GNNs when reasoning algorithmically, for some combinatorial applications, we require the algorithmic performance to always remain perfect—a trait known as *strong generalization* (Li et al., 2020). Strong generalization was demonstrated to be possible. That is, neural execution engines (NEEs) (Yan et al., 2020) are capable of maintaining 100% accuracy on various combinatorial tasks by leveraging several low-level constructs, learning individual primitive units of computation, such as addition, multiplication, or argmax, in isolation. Moreover, they employ explicit masking inductive biases and binary representations of inputs. Here, the focus is less on learning the algorithm itself—the dataflow between the computation units is provided in a hard-coded way, allowing for zero-shot transfer of units between related algorithms (such as Dijkstra et al. (1959) and Prim (1957), which have an identical control flow backbone).

3.3.3 Hierarchy of reasoners

From the preceding discussion, a natural hierarchy of neural algorithmic reasoning approaches emerges—one that represents a convenient analog to the hierarchy of programming languages.

Algo-level approaches (Xu et al., 2019b; Joshi et al., 2020; Tang et al., 2020; Awasthi et al., 2021) focus on learning entire algorithms, end-to-end, from inputs to outputs. Learning end-to-end is the highest level of abstraction, which admits easier theoretical analysis, but may suffer in generalization performance. By analogy, consider high-level programming languages (such as Python) that allow for simpler specification of programs at the expense of computational performance.

Step-level approaches (Veličković et al., 2020b,a; Georgiev and Lió, 2020; Deac et al., 2020a; Strathmann et al., 2021) focus on learning atomic steps of algorithms, through strong intermediate supervision. This level allows for maintaining an end-to-end structure while significantly boosting extrapolation performance and reducing the sample complexity, at the expense of strongly regularizing the model and requiring additional training data. By analogy, consider medium-level programming languages (such as C++), which attempt to model a "middle-ground" of providing useful high-level constructs while still allowing direct access to the internals of the underlying machine.

Unit-level approaches (Yan et al., 2020) focus on strongly learning primitive units of computation, then specifying hard-coded or nonparametric means of combining such units. Such approaches enable perfect generalization but are no longer focused on full-algorithmic representations. That is, algorithms are usually specified by manually

composing units, making the method no longer end-to-end. By analogy, consider low-level programming languages (such as Assembly) that are perfectly aligned to the underlying hardware and offer maximal performance but require carefully designing every subroutine with lots of repetition.

3.3.4 Reasoning on natural inputs

Until now, we have focused on methodologies that allowed for GNNs to strongly reason out-of-distribution, purely by more faithfully imitating existing classical algorithms. Imitation is an excellent way to benchmark GNN architectures for their reasoning capacity. In theory, it allows for infinite amounts of training or testing data of various distributions, and the fact that the underlying algorithm is known means that extrapolation can be rigorously defined.[9] However, an obvious question arises: if all we are doing is imitating a classical algorithm, why not just apply the algorithm?

There are many potential applications of algorithmic reasoning which may provide answers to this question in principle.[10] However, one particularly appealing direction for CO has already emerged—algorithmic learning executors allows us to generalize these classical combinatorial reasoners to natural inputs. We will thoroughly elaborate on this here.

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. For an example of why this is at timeless odds with the way such algorithms are designed, we will look back to a 1955 study by Harris and Ross (1955), which is among the first to introduce the *maximum flow* problem, before the seminal work of Ford and Fulkerson (1956), and Dinic (1970), both of which present algorithms for solving it.

In line with the Cold War's contemporary issues, Harris and Ross studied the Soviet railway lines' bottleneck properties. They analyzed the rail network as a graph with edges representing railway links with scalar *capacities*, corresponding to the train traffic flow rate that the railway link may support. The authors used this representation as a tool to search for the *bottleneck capacity*—identifying links that would be the most effective targets for the aerial attack to disrupt the capacity maximally. Subsequent analyses have shown that this problem can be related to the *minimum cut* problem on graphs and can be shown equivalent to finding a maximal flow through the network; this follows directly from the subsequently proven *max-flow min-cut theorem* (Ford and Fulkerson, 2015). This problem inspired a very fruitful stream of novel combinatorial algorithms and data structures (Ford and Fulkerson, 1956; Edmonds and Karp, 1972; Dinic, 1970; Sleator and Tarjan, 1983; Goldberg and Tarjan, 1988), with applications stretching far beyond the original intent.

9. In principle, any function could be a correct extrapolant if the underlying target is not known.

10. Perhaps a more "direct" application is the ability to *discover* novel algorithms. This is potentially quite promising, as most classical algorithms were constructed with a single-threaded CPU model in mind, and many of their computations may be amenable to more efficient execution on a GPU. There certainly exist preliminary signs of potential: Li et al. (2020) had detected data-driven sorting procedures that seem to improve on quicksort, and Veličković et al. (2020a) indicate, on small examples, that they were able to generalise the operations of the disjoint-set union data structure in a GPU-friendly way.

However, throughout their writeup, Harris and Ross remain persistently mindful of one crucial shortcoming of their proposal: the need to attach a single, scalar capacity to an entire railway link necessarily ignores a potential wealth of nuanced information from the underlying system. Quoting verbatim just one such instance:

> *"The evaluation of both railway system and individual track capacities is, to a considerable extent, an art. The authors know of no tested mathematical model or formula that includes all of the variations and imponderables that must be weighed.* Even when the individual has been closely associated with the particular territory he is evaluating, the final answer, however accurate, is largely one of judgment and experience."*

In many ways, this problem continues to plague applications of classical CO algorithms, being able to satisfy their 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. On the other hand, the data we need to apply the algorithm may to be only partially observable, and this can often render the algorithm completely inapplicable. Both points should be recognized as important issues within the CO as well as operations research communities.

Both of these issues sound like fertile ground for neural networks. Their capabilities, both as a replacement for human feature engineering and a powerful processor of raw data, are highly suggestive of their potential applicability. However, here we hit a key obstacle. 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[11] 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 that the inputs are free of noise and estimated correctly. In contrast, neural networks derive great flexibility from their latent representations, that are inherently high-dimensional,[12] if any component of the neural representation ends up poorly predicted, other components are still able to step in and compensate. This is partly what enabled neural networks' emergence as a flexible tool for raw data processing. If there is insufficient data to learn how to compress it into scalar values meaningfully, this may make the ultimate results of applying combinatorial algorithms on them suboptimal.

Mindful of the above, we can identify that the latest advances in neural algorithmic reasoning could lend a remarkably elegant pipeline for reasoning on natural inputs. The

11. Proposals for perceptive black-box CO solvers have also emerged outside the realm of end-to-end learning; for example, Brouard et al. (2020) demonstrate an effective perceptive combinatorial solver by leveraging a convex formulation of graphical models.

12. There is a caveat that allows *some* classical combinatorial algorithms to escape this bottleneck; namely, if they are *designed* to operate over high-dimensional latent representations, one may just apply them out-of-the-box to the latent representations of neural networks. A classical example is k-means clustering: this insight lead Wilder et al. (2019) to propose the powerful ClusterNet model.

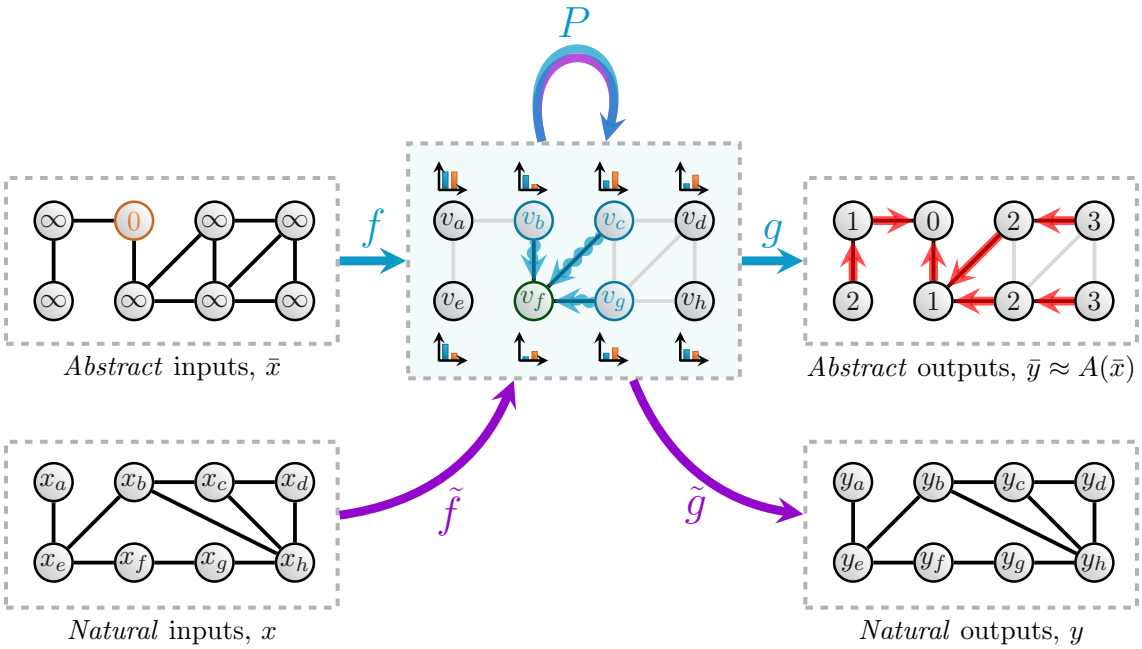


Figure 7: The proposed algorithmic reasoning blueprint. First, an algorithmic reasoner is trained in the encode-process-decode fashion, learning a function $g(P(f(\bar{x}))) \approx A(\bar{x})$, for a target combinatorial algorithm A; in this case, A is breadth-first search. Once trained, the processor network P is frozen and stitched into a pipeline over natural inputs—with new encoder and decoder $\tilde{f}$ and $\tilde{g}$. This provides an end-to-end differentiable function that has no explicit information loss, while retaining alignment with BFS.

power comes from using the aforementioned encode-process-decode framework. 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 polynomial-time 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.

Such a processor thus seems to be a perfect component in a neural end-to-end pipeline which goes straight from raw inputs to general outputs. The general procedure for applying an algorithm A (which admits abstract inputs $\bar{x}$) to raw inputs x is as follows (see Figure 7):

1. Learn an algorithmic reasoner for A, on synthetically generated 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. Set up appropriate encoder and decoder neural networks, $\tilde{f}$ and $\tilde{g}$, to process raw data and produce desirable outputs.[13] The encoder should produce embeddings that correspond to the input dimension of P, while the decoder should operate over input embeddings that correspond to the output dimension of P.

3. Swap out f and g for $\tilde{f}$ and $\tilde{g}$, and learn their parameters by gradient descent on any differentiable loss function that compares $\tilde{g}(P(\tilde{f}(x)))$ to ground-truth outputs, y. The parameters of P should be kept *frozen* throughout this process.

Therefore, algorithmic reasoning presents a strong approach—through pre-trained processors[14]—to reasoning over natural inputs. The raw encoder function $\tilde{f}$ has the potential to eliminate the human feature engineer from the CO/OR 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.

Value Iteration (VI) represents one of the most prominent model-based planning algorithms, which is guaranteed to converge to an optimal RL policy. However, it requires the underlying Markov decision process to be discrete, fixed, and completely known—all requirements that are hardly satisfied in most settings of interest to deep RL. Its appeal had inspired prior work on designing neural networks that algorithmically align with VI in certain special cases, namely, in grid-worlds[15] VI aligns with convolution. This yielded the Value Iteration Network architecture (Tamar et al., 2016, VIN), which carefully leveraged convolutions and weight sharing to demonstrate superior generalization performance compared to standard CNN agents. While extensions to graph-based environments using GNNs have been made (Niu et al., 2018), the above constraints on the MDP remained.

In the XLVIN architecture, Deac et al. (2020b) have surpassed these limitations by following precisely the algorithmic reasoning blueprint above. They pre-trained an algorithmic executor for VI on several synthetic and known MDPs, then deployed it over a local neighborhood of the current state, derived using self-supervised learning.

The representations produced by this VI executor substantially improved a corresponding model-free RL baseline, especially in terms of data efficiency. Additionally, the model performed strongly in the low-data regime against ATreeC (Farquhar et al., 2017), which resorted to predicting scalar values in every node of the inferred local MDP, so that VI-style rules can be directly applied exactly according to the predictions above. Even over challenging RL environments such as Atari, neurally learned algorithmic cores have been proven a viable way of applying classical combinatorial algorithms to natural inputs in a way that can

13. In the case where the desired output is exactly the output of the algorithm, one may set $\tilde{g} = g$ and re-use the decoder.

14. While presenting an earlier version of our work, a very important point was raised by Max Welling: if our aim is to encode a high-dimensional algorithmic solver within P, why not just set its weights manually to match the algorithm's steps? While this would certainly make P trivially extrapolate, it is our belief that it would be very tricky to manually initialize it in a way that *robustly* and *diversely* uses all the dimensions of its latent input. And if P only sparsely uses its latent input, we would be faced with yet another algorithmic bottleneck, limiting data efficiency. That being said, deterministic distillation of algorithms into robust high-dimensional processor networks is a potentially exciting area for future work.

15. Note that this does not ameliorate the requirements listed above. Assuming an environment is a grid-world places strong assumptions on the underlying MDP.

surpass even a hard-coded hybrid pipeline. This is a first-view account of the potential of neural algorithmic reasoning in the real world and, given that XLVIN is only one manner in which this blueprint may see the application, we anticipate that it paves the way for many more practical applications of CO.

4. Limitations and Research Directions

In the following, we give an overview of works that quantify the limitations of GNNs and the resulting implications for their use in CO. Moreover, we provide directions for further research.

4.1 Limitations

In the following, we survey known limitations of GNN approaches to CO.

Expressivity of GNNs Recently, different works explore the limitations of GNNs (Xu et al., 2019a; Morris et al., 2019). Specifically, Morris et al. (2019) show that any GNN architecture's power to distinguish non-isomorphic graphs is upper-bounded by the 1-dimensional Weisfeiler-Leman algorithm (Weisfeiler and Leman, 1968), a well-known polynomial-time heuristic for the graph isomorphism problem. The heuristic is well-understood and is known to have many shortcomings (Arvind et al., 2015), such as not being able to detect cyclic information or distinguish between non-isomorphic bipartite graphs. These shortcomings have direct implications for CO applications, as they imply the existence of pairs of non-equal MIP instances that no GNN architecture can distinguish. This inspired a large body of research on stronger variants of GNNs (Chen et al., 2019; Morris et al., 2019; Maron et al., 2019a,b; Morris et al., 2020; Murphy et al., 2019a,b) that provably overcome these limitations. However, such models typically do not scale to large graphs, making their usage in CO prohibitive. Alternatively, recent works (Sato et al., 2020; Abboud et al., 2020) indicate that randomly initialized node features can help boost expressivity of GNNs, although the impact of such an approach on generalization remains unclear.

Generalization of GNNs To deploy successful supervised machine learning models for CO, understanding generalization (out-of-training-set performance) is crucial. Garg et al. (2020) prove generalization bounds for a large class of GNNs that depend mainly on the maximum degree of the graphs, the number of layers, width, and the norms of the learned parameter matrices. Importantly, these bounds strongly depend on the sparsity of the input, which suggests that GNN's generalization ability might worsen the denser the graphs get.

Approximation and computational power As explained in Section 3.1, GNNs are often designed as (part of) a direct heuristic for CO tasks. Therefore, it is natural to ask what is the best *approximation ratio* achievable on various problems. By transferring results from distributed local algorithms (Suomela, 2013), Sato et al. (2019) show that the best approximation ratio achievable by a large class of GNNs on the minimum vertex cover problem is 2, which is suboptimal (Karakostas, 2005). They also show analogous suboptimality results regarding the minimum dominated set problem and the maximum matching problem. Regarding computability, Loukas (2020) prove that some GNNs can be

too small to compute some properties of graphs, such as finding their diameter or a minimum spanning tree and give minimum depth and width requirements for such tasks.

Large inference cost In some machine learning applications for CO, the inference might be repeated thousands of times to minimize the *wall-clock time* being a core objective. A typical example is repeated decision-making within a CO solver, e.g., branching. In this common scenario, making worse decisions fast might lead to better overall solving times than good decisions slowly. The low-degree polynomial complexity of GNN inference might be insufficient to be competitive against simpler models in this setting. A recent work (Gupta et al., 2020) suggests a hybrid approach in one of these scenarios by running a full-fledged GNN once and using a suitably trained MLP to continue making decisions using the embedding computed by the GNN with additional features.

Data limitations in CO Making the common assumption that the complexity classes NP and co-NP are not equal, Yehuda et al. (2020) show that any polynomial-time sample generator for NP-hard problems samples from an easier sub-problem. Under some circumstances, these sampled problems may even be *trivially classifiable*; for example, a classifier only checks the value of one input feature. This indicates that the observed performance metrics of current supervised approaches for intractable CO tasks may be over-inflated. However, it remains unclear how these results translate into practice, as real-world instances of CO problems are rarely worst-case ones.

4.2 Proposed New Directions

To stimulate further research, we propose the following key challenges and extensions.

Understanding the trade-off in scalability, expressivity, and generalization As outlined in the previous subsection, current GNN architectures might miss crucial structural patterns in the data, while more expressive approaches do not scale to large-scale inputs. What is more, decisions inside CO 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 patterns remains an open challenge. However, increased expressiveness might negatively impact generalization. Nowadays, most of the supervised approaches do not give meaningful predictive performance when evaluated on out-of-training-distribution samples. Even evaluating trained models on slightly larger graph instances often leads to a significant drop in performance. 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 there 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, which are very common in bioinformatics, and explicitly supporting recursive primitives, for which any existing GNN executor would eventually run out of representational capacity.

Perceptive CO Significant strides have already been made to use GNNs to strengthen abstractified CO pipelines. Further efforts are needed to support combinatorial reasoning over real-world inputs as most CO problems are ultimately 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 cases, exactly.

Building a generic implementation framework for GNNs for CO Although implementation frameworks for GNNs have now emerged, it is still cumbersome to integrate GNN and machine learning into state-of-the-art solvers for the practitioners. Hence, developing a kind of modeling language for integrating ML methods that abstract from technical details remains an open challenge and is key for adopting machine learning and GNN approaches in the real world, some the early attempts are discussed in the next section.

5. Implementation Frameworks

Nowadays, there are several well-documented, open-source libraries for implementing custom GNN architectures, providing a large set of readily available models from the literature. The most used such libraries are PyTorch Geometric (Fey and Lenssen, 2019) and Deep Graph Library (Wang et al., 2019). Conversely, libraries to simplify the usage of machine learning in CO have also been developed. OR-Gym (Hubbs et al., 2020) and OpenGraphGym (Zheng et al., 2020) are libraries designed to facilitate the learning of heuristics for CO problems in a similar interface to the popular OpenAI Gym library (Brockman et al., 2016). In contrast, MIPLearn (Xavier and Qiu, 2020) is a library that facilitates the learning of configuration parameters for CO solvers. Ecole (Prouvost et al., 2020) offers a general, extensible framework for implementing and evaluating machine learning-enhanced CO. It is also based on OpenAI Gym, and it exposes several essential decision tasks arising in general-purpose CO solvers—such as SCIP (Gamrath et al., 2020)—as control problems

over MDPs. Finally, SeaPearl (Chalumeau et al., 2021) is a constraint programming solver guided by reinforcement learning and which uses GNNs for representing training instances.

6. Conclusions

We gave an overview of the recent applications of GNNs for CO. To that end, we gave a concise introduction to CO, the different machine learning regimes, and GNNs. Most importantly, 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. We discussed shortcomings and research directions regarding the application of GNNs to CO. Finally, we identified a set of critical challenges to stimulate future research and advance the emerging field. We hope that our survey presents a useful handbook of graph representation learning methods, perspectives, and limitations for CO, operations research, and machine learning practitioners alike and that its insights and principles will be helpful in spurring novel research results and future avenues.

References

E. Aarts and J. K. Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

R Abboud, İ. İ. Ceylan, M. Grohe, and Lukasiewicz T. The surprising power of graph neural networks with random node initialization. *CoRR*, abs/2010.01179, 2020.

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

V. Arvind, J. Köbler, G. Rattan, and O. Verbitsky. On the power of color refinement. In *International Symposium on Fundamentals of Computation Theory*, pages 339–350, 2015.

P. Awasthi, A. Das, and S. Gollapudi. Beyond GNNs: A sample efficient architecture for graph problems, 2021. URL `https://openreview.net/forum?id=Px7xIKHjmMS`.

R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

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

I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017.

Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning*, volume 382, pages 41–48, 2009.

Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

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

D. Bertsimas and J.N. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

I. Boussaïd, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information sciences*, 237:82–117, 2013.

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

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

C. Brouard, S. de Givry, and T. Schiex. Pushing data into cp models using graphical model learning and solving. In *International Conference on Principles and Practice of Constraint Programming*, pages 811–827. Springer, 2020.

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and deep locally connected networks on graphs. In *International Conference on Learning Representation*, 2014.

C. Cameron, R. Chen, J. S. Hartford, and K. Leyton-Brown. Predicting propositional satisfiability via end-to-end learning. In *AAAI Conference on Artificial Intelligence*, pages 3324–3331, 2020.

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

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.

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

I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *CoRR*, abs/2005.03675, 2020.

Z. Chen, S. Villar, L. Chen, and J. Bruna. On the equivalence between graph isomorphism testing and function approximation with GNNs. In *Advances in Neural Information Processing Systems*, pages 15868–15876, 2019.

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

G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. Principal neighbourhood aggregation for graph nets. *CoRR*, abs/2004.05718, 2020.

G. A. Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6): 791–812, 1958.

H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

A. Deac, P.-L. Bacon, and J. Tang. Graph neural induction of value iteration. *CoRR*, abs/2009.12604, 2020a.

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

M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the TSP by policy gradient. In *International conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181, 2018.

E. W. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

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 Conference on Artificial Intelligence*, 2020.

E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

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.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

M. Dror, G. Laporte, and P. Trudeau. Vehicle routing with split deliveries. *Discrete Applied Mathematics*, 50(3):239–254, 1994.

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.

D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, pages 2224–2232, 2015.

K. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson. TreeQN and ATreeC: Differentiable tree planning for deep reinforcement learning. *CoRR*, abs/1710.11417, 2017.

M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. *CoRR*, abs/1903.02428, 2019.

M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege. Deep graph matching consensus. In *International Conference on Learning Representations*, 2020.

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

L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

L. R. Ford and D. R. Fulkerson. *Flows in networks*. Princeton University Press, 2015.

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.

K. Freivalds, E. Ozoliņš, and Šostaks. Neural shuffle-exchange networks-sequence processing in O(n log n) time. In *Advances in Neural Information Processing Systems*, pages 6626–6637, 2019.

A. Galler, B and M. J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

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. E. 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.

V. K. Garg, S. Jegelka, and T. S. Jaakkola. Generalization and representational limits of graph neural networks. *CoRR*, abs/2002.06157, 2020.

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 15554–15566, 2019.

D. Georgiev and P. Lió. Neural bipartite matching. *CoRR*, abs/2005.11304, 2020.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.

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.

F. Glover and M. Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

F. W. Glover and G. A. Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

A. Graves, G. Wayne, and I. Danihelka. Neural Turing machines. *CoRR*, abs/1410.5401, 2014.

A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. Gómez Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. Badia Puigdomènech, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

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.

A. Guzman-Rivera, D. Batra, and P. Kohli. Multiple choice learning: Learning to produce multiple structured outputs. In *Advances in Neural Information Processing Systems*, pages 1808–1816, 2012.

W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1025–1035, 2017.

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.

T. E. Harris and F. S. Ross. Fundamentals of a method for evaluating rail net capacities. Technical report, RAND Coperation, Santa Monica, CA, 1955.

C. D. Hubbs, H. D. Perez, O. Sarwar, N. V. Sahinidis, I. E. Grossmann, and J. M. Wassick. OR-Gym: A reinforcement learning library for operations research problems. *CoRR*, abs/2008.06319, 2020.

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

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

N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

L. Kaiser and I. Sutskever. Neural GPUs learn algorithms. *CoRR*, abs/1511.08228, 2015.

G. Karakostas. A better approximation ratio for the vertex cover problem. In *International Colloquium on Automata, Languages, and Programming*, pages 1043–1050. Springer, 2005.

N. Karalias and A. Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. In *Advances in Neural Information Processing Systems*, 2020.

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

E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representation*, 2017.

D. B. Kireev. Chemnet: A novel neural network based method for graph/property mapping. *Journal of Chemical Information and Computer Sciences*, 35(2):175–180, 1995.

P. Knöbelreiter, C. Reinbacher, A. Shekhovtsov, and T. Pock. End-to-end training of hybrid cnn-crf models for stereo. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2339–2348, 2017.

W. Kool, H. Van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

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

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

K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *CoRR*, abs/1511.06392, 2015.

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 *Advances in Neural Information Processing Systems*, 2020.

L. C. Lamb, A. S. d'Avila Garcez, M. Gori, M. O. R. Prates, P. H. C. Avelar, and M. Y. Vardi. Graph neural networks meet neural-symbolic computing: A survey and perspective. In *International Joint Conference on Artificial Intelligence*, pages 4877–4884, 2020.

G. Lederman, M. N. Rabe, and S. A. Seshia. Learning heuristics for automated reasoning through deep reinforcement learning. In *International Conference on Learning Representations*, 2020.

Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International Conference on Machine Learning*, pages 3835–3845, 2019.

Y. Li, F. Gimeno, P. Kohli, and O. Vinyals. Strong generalization and efficiency in neural programs. *CoRR*, abs/2007.03629, 2020.

Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 537–546, 2018.

D. Liu, A. Lodi, and M. Tanneau. Learning chordal extensions. *CoRR*, abs/1910.07600, 2019.

A. Lodi and G. Zarpellon. On learning and branching: A survey. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, 25(2):207–236, July 2017.

A. Loukas. What graph neural networks cannot learn: Depth vs width. In *International Conference on Learning Representations*, 2020.

K. Lu and M. P. Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations*, 2020.

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.

A. Madsen and A. Rosenberg Johansen. Neural arithmetic units. In *International Conference on Learning Representations*, 2020.

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

H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably powerful graph networks. In *Advances in Neural Information Processing Systems*, pages 2153–2164, 2019a.

H. Maron, H. Ben-Hamu, N. Shamir, and Y. Lipman. Invariant and equivariant graph networks. In *International Conference on Learning Representations*, 2019b.

N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev. Reinforcement learning for combinatorial optimization: A survey. *CoRR*, abs/2003.03600, 2020.

C. Merkwirth and T. Lengauer. Automatic generation of complementary descriptors with molecular graph networks. *Journal of Chemical Information and Modeling*, 45(5):1159–1168, 2005.

A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, et al. Chip placement with deep reinforcement learning. *CoRR*, abs/2004.10746, 2020.

N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2012.

F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 5425–5434, 2017.

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 *Conference on Artificial Intelligence*, pages 4602–4609, 2019.

C. Morris, G. Rattan, and P. Mutzel. Weisfeiler and Leman go sparse: Towards higher-order graph embeddings. In *Advances in Neural Information Processing Systems*, 2020.

R. L. Murphy, B. Srinivasan, V. A. Rao, and B. Ribeiro. Relational pooling for graph representations. In *International Conference on Machine Learning*, pages 4663–4673, 2019a.

R. L. Murphy, B. Srinivasan, V. A. Rao, and B. Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. In *International Conference on Learning Representations*, 2019b.

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

M. Nazari, A. Oroojlooy, M. Takáč, and L. V. Snyder. Reinforcement learning for solving the vehicle routing problem. In *International Conference on Neural Information Processing Systems*, pages 9861–9871, 2018.

S. Niu, S. Chen, H. Guo, C. Targonski, M. Smith, and J. Kovačević. Generalized value iteration networks: Life beyond lattices. In *AAAI Conference on Artificial Intelligence*, 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. IEEE, 2018.

R. B. Palm, U. Paquet, and O. Winther. Recurrent relational networks. *arXiv preprint arXiv:1711.08028*, 2017.

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

A. S. Polydoros and L. Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

M. 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 *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4731–4738, 2019.

R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

A. Pritzel, B. Uria, S. Srinivasan, A. P. Badia, O. Vinyals, D. Hassabis, D. Wierstra, and C. Blundell. Neural episodic control. In *International Conference on Machine Learning*, pages 2827–2836, 2017.

A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. *CoRR*, abs/2011.06069, 2020.

S. Reed and N. De Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *National Conference on Artificial Intelligence*, pages 362–367, 1994.

O. Richter and R. Wattenhofer. Normalized attention without probability cage. *CoRR*, abs/2005.09561, 2020.

Stephane Ross. *Interactive Learning for Sequential Decisions and Predictions*. PhD thesis, Carnegie Mellon University, 2013.

F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

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.

A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap. A simple neural network module for relational reasoning. In *Advances in Neural Information Processing Systems*, pages 4967–4976, 2017.

A. Santoro, R. Faulkner, D. Raposo, J. Rae, M. Chrzanowski, T. Weber, D. Wierstra, O. Vinyals, R. Pascanu, and T. Lillicrap. Relational recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 7299–7310, 2018.

R. Sato, M. Yamada, and H. Kashima. Approximation ratios of graph neural networks for combinatorial problems. In *Advances in Neural Information Processing Systems*, pages 4083–4092, 2019.

R. Sato, M. Yamada, and H. Kashima. Random features strengthen graph neural networks. *CoRR*, abs/2002.03155, 2020.

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

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

D. Selsam and N. Bjørner. NeuroCore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR*, abs/1903.04671, 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 *International Conference on Learning Representations*, 2019.

S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

M. Shanahan, K. Nikiforou, A. Creswell, C. Kaplanis, D. G. T. Barrett, and M. Garnelo. An explicitly relational neural network architecture. In *International Conference on Machine Learning*, pages 8593–8603, 2020.

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.

D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

Kate A Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(2):714–35, 1997.

H. Strathmann, M. Barekatain, C. Blundell, and P. Veličković. Persistent message passing. *CoRR*, abs/2103.01043, 2021.

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.

J. Suomela. Survey of local algorithms. *ACM Computing Surveys*, 45(2):24:1–24:40, 2013.

A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel. Value iteration networks. *Advances in Neural Information Processing Systems*, 29:2154–2162, 2016.

H. Tang, Z. Huang, J. Gu, B.-L. Lu, and H. Su. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *Advances in Neural Information Processing Systems*, 33, 2020.

R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

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.

P. Toth and S. Vigo. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

A. Trask, F. Hill, S. E. Reed, J. Rae, C. Dyer, and P. Blunsom. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pages 8035–8044, 2018.

P. Vaezipoor, G. Lederman, Y. Wu, C. J. Maddison, R. Grosse, E. Lee, S. A. Seshia, and F. Bacchus. Learning branching heuristics for propositional model counting. *CoRR*, abs/2007.03204, 2020.

P. J. M. Van Laarhoven and E. H. L. Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

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

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

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

P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020b.

N. Vesselinova, R. Steinert, D. F. Perez-Ramirez, and M. Boman. Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 8: 120388–120416, 2020.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2020.

L. Vrček, P. Veličković, and M. Šikić. A step towards neural genome assembly. *CoRR*, abs/2011.05013, 2020.

M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *CoRR*, abs/1909.01315, 2019.

P.-W. Wang, P. Donti, B. Wilder, and Z. Kolter. SATnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554, 2019.

B. Weisfeiler and A. Leman. The reduction of a graph to canonical form and the algebra which appears therein. *NTI, Series*, 2(9):12–16, 1968.

B. Wilder, E. Ewing, B. Dilkina, and M. Tambe. End to end learning and optimization on graphs. In *Advances in Neural Information Processing Systems*, pages 4674–4685, 2019.

R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

A. S. Xavier and F. Qiu. MIPLearn, 2020. URL `https://anl-ceeesa.github.io/MIPLearn`.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019a.

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

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.

Y. Yan, K. Swersky, D. Koutra, P. Ranganathan, and M. Hashemi. Neural execution engines. In *Advances in Neural Information Processing*, 2020.

Y. Yang and A. B. Whinston. A survey on reinforcement learning for combinatorial optimization. *CoRR*, abs/2008.12248, 2020.

Y. Yang, T. Liu, Y. Wang, J. Zhou, Q. Gan, Z. Wei, Z. Zhang, Z. Huang, and D. Wipf. Graph neural networks inspired by classical iterative algorithms. *CoRR*, abs/2103.06064, 2021.

G. Yehuda, M. Gabel, and A. Schuster. It's not what machines can learn, it's what we cannot teach. *CoRR*, abs/2002.09398, 2020.

R. Yolcu and B. Póczos. Learning local search heuristics for boolean satisfiability. In *Advances in Neural Information Processing Systems*, pages 7992–8003, 2019.

W. Zaremba and I. Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *International Joint Conference on Artificial Intelligence*, pages 1114–1120, 1995.

W. Zheng, D. Wang, and F. Song. OpenGraphGym: A parallel reinforcement learning framework for graph optimization problems. In *International Conference on Computational Science*, pages 439–452. Springer, 2020.

J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.