ACM DIGITAL LIBRARY
acm open

Latest updates: https://dl.acm.org/doi/10.5555/3648699.3648829

RESEARCH-ARTICLE

# Combinatorial optimization and reasoning with graph neural networks

**QUENTIN CAPPART**, Polytechnique Montral, Montreal, QC, Canada

**DIDIER CHÉTELAT**, Polytechnique Montral, Montreal, QC, Canada

**ELIAS BOUTROS KHALIL**, University of Toronto, Toronto, ON, Canada

**ANDREA LODI**

**CHRISTOPHER MORRIS**, RWTH Aachen University, Aachen, Nordrhein-Westfalen, Germany

**PETAR VELIČKOVIĆ**, DeepMind Technologies Limited, London, U.K.

**Open Access Support** provided by:

**University of Toronto**

**Polytechnique Montral**

**DeepMind Technologies Limited**

**RWTH Aachen University**

**PDF Download**
**3648699.3648829.pdf**
**23 February 2026**
**Total Citations:** 0
**Total Downloads:** 487

**Published:** 01 January 2023
**Accepted:** 01 April 2023
**Revised:** 01 January 2023
**Received:** 01 April 2021

**Citation in BibTeX format**

Journal of Machine Learning Research 24 (2023) 1-61 Submitted 4/21; Revised 1/23; Published 4/23

# 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@CORNELL.EDU
*Jacobs Technion-Cornell Institute*
*Cornell Tech and Technion - IIT*
*New York, USA*

**Christopher Morris** MORRIS@CS.RWTH-AACHEN.DE
*Department of Computer Science*
*RWTH Aachen University*
*Aachen, Germany*

**Petar Veličković** PETARV@DEEPMIND.COM
*DeepMind*
*London, UK*

**Editor:** Francis Bach

## 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 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 optimization and machine learning researchers.

**Keywords:** Combinatorial optimization, graph neural networks, reasoning

©2023 Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, Petar Veličković.

License: CC-BY 4.0, see https://creativecommons.org/licenses/by/4.0/. Attribution requirements are provided at http://jmlr.org/papers/v24/21-0449.html.

## 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, 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 must solve problem instances with specific characteristics or patterns. For example, a trucking company may solve vehicle routing instances for the same city daily, with only slight differences across instances 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; Gasse et al., 2022). The promise is that one can develop faster algorithms for practical cases by exploiting common patterns in the given instances.

Due to the discrete nature of most CO problems and the prevalence of network data in the real world, graphs are a central object of study in the CO field. For example, well-known and relevant problems such as the Traveling Salesperson problem (TSP) and other vehicle routing problems naturally induce a graph structure. In fact, from the 21 NP-complete problems identified by Karp (1972), ten are decision versions of graph optimization problems. Most other ones, such as the set covering problem, can also be modeled over graphs. Moreover, the interaction between variables and constraints in combinatorial 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 exhibit symmetries, 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 are large and usually sparse, especially those arising from the real world. 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. They require a large amount of training data to optimize the model's parameters. 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 generalize beyond its training data, e.g., transferring to instances of different sizes.

Overall, there is a trade-off between scalability, expressivity, and generalization, 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's *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, need 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* or *out-of-distribution* instances.

### 1.2 How Do GNNs Address These Challenges?

*Graph neural networks* (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. 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 by parameterizing this aggregation step. 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 (Morris et al., 2021).

Although GNNs have clear limitations, which we will also explore and outline, they have already proven useful in 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. (2021), 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 node features in a (permutation-)invariant way. 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 only scratches the surface of the innovations that can be enabled by a careful synergy of GNNs and CO. 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 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 railways network analysis (Harris and Ross, 1955), but remained out of the spotlight of theoretical computer science that 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 3.3.3.

### 1.4 Present Work

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

1. We provide a comprehensive, structured overview of the application of GNNs to the CO setting for both heuristic and exact algorithms.
2. We survey recent progress in using GNN-based end-to-end algorithmic reasoners.
3. We highlight the shortcomings of GNNs in the context of CO and provide guidelines and recommendations on how to tackle them.
4. We provide a list of open research directions to stimulate future research.

We believe that reaping the benefits of GNNs for CO is a promising research direction. This survey is intended to provide the required material for readers eager to discover this field. On the other hand, we highlight that this survey should not be considered as a recommendation that GNNs is the best way to solve CO problems. There are clear limitations and fundamental challenges to tackle. Besides, the results obtained are currently below what can be achieved by traditional CO solvers in most situations. Such limitations are discussed throughout the paper and summarized in Section 4.

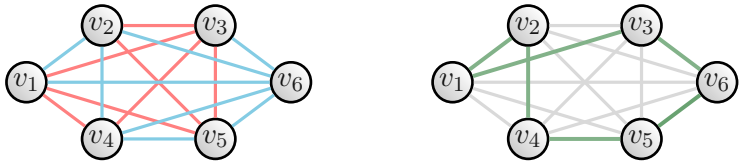
### 1.5 Related Work

The following briefly reviews 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); Merkwirth and Lengauer (2005); Scarselli et al. (2009); Sperduti and Starita (1997). Aligned with the field's recent rise in popularity, many surveys exist on recent advances in GNN techniques. Some of the most recent ones include Chami et al. (2020); Wu et al. (2019); Zhou et al. (2020).

**Continuous Formulations** The discrete nature of CO problems makes standard continuous optimization tools unavailable, such as first- and second-order gradient methods. However, many problems admit alternative reformulations as non-convex continuous optimization problems over graphs. Such problems include graph partitioning, maximum cut, minimum vertex cover, maximum independent set, and maximum clique problems. Some early work at the intersection of machine learning and combinatorial optimization involves reinterpreting these continuous optimization problems as energy-based training of Hopfield neural networks or self-organizing maps, such as in the work of Hopfield and Tank (1985), Durbin and Willshaw (1987), Ramanujam and Sadayappan (1995) and Gold et al. (1996). Although not using GNNs, these works use graphs as a central object. They can be seen as foreshadowing various GNN-based differentiable proxy loss approaches that we summarize in Section 3.1.1.

**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, as described in the previous section. 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) focus 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 end-to-end learning setups and paradigms, making representation learning—and GNNs in particular—a secondary topic. Moreover, the surveys by Mazyavkina et al. (2021); 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) give a high-level overview of the application of GNNs in various reasoning tasks, missing out on the most recent developments, e.g., the algorithmic reasoning direction we study in detail here.




(a) Instance. (b) Optimal solution.

Figure 1: 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.

### 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 CO context. 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 intending to stimulate 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 $\Sigma$ 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* $(\Omega, F, w)$*, where* $\Omega$ *is a* finite *set,* $F \subseteq 2^{\Omega}$ *is the set of* feasible *solutions,* $c\colon 2^{\Omega} \to \mathbb{R}$ *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 $c$ 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 1 for an illustration of an instance of the TSP problem and its optimal solution. The corresponding decision problem asks whether a cycle exists along the edges of a graph with cost $\leq k$ that visits each node exactly once.

**Example 1 (Traveling Salesperson Problem)**
*Input: A complete directed graph* $G$*, i.e.,* $E(G) = \{(u, v) \mid u, v \in V(G)\}$*, with edge costs* $w\colon E(G) \to \mathbb{R}$*.*
*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-hard (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 algorithms (and associated 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

We describe common modeling and algorithmic frameworks for CO problems in the following. More precisely, the next three sections describe the modeling approaches: integer programming, SAT, and constraint satisfaction/optimization. Finally, Section 2.3.4 partitions the algorithmic frameworks into three categories. Note that this section is not an exhaustive list of optimization approaches but serves as an introduction to the main frameworks.

#### 2.3.1 Integer linear programs and mixed-integer programs

First, we define 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 $\mathbb{R}^{m \times n}$, and $b$ and $c$ are vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively.*

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

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

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*, i.e., 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-bound algorithms, see Section 2.3.4 for details. By dropping the integrality constraints, we again obtain an instance of an LP, which we call *relaxation*. Solving the LP relaxation of an ILP provides a valid lower bound on the optimal solution of the problem, i.e., an optimistic approximation, and the quality of such an approximation is largely responsible of the effectiveness of the search scheme.

**Example 2** *We provide an ILP that encodes all feasible solutions of the TSP and selects the optimal one due to the objective function. 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 traveling from city $i$ to city $j$, $i \neq j$. Then, the TSP can be written as the following ILP:*[3]

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

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.
3. Technically, the presented TSP model is for the *asymmetric* version, where the costs $w_{ij}$ and $w_{ji}$ might be different. Such a TSP version is represented in a directed graph. Instead, the version in Figure 1 is *symmetric*, i.e., $w_{ij} = w_{ji}$, and it is represented on an undirected graph.

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

In practice, one often faces problems consisting 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 $\mathbb{R}^n$ that minimizes $c^T x$ over the *feasible set*

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

Here, $n$ is the number of variables we optimize, out of which $p$ must 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: A propositional logic formula $\varphi$ with variable set $V$.*
*Output:* Yes*, if there exists a variable assignment $A\colon V \to \{true, false\}$ such that the formula $\varphi$ evaluates to true;* No*, otherwise.*

The SAT problem was the first one to be shown to be NP-complete (Cook, 1971). However, modern solvers routinely solve industrial-scale instances in practice (Prasad et al., 2005). Despite the simplicity of its formalization, SAT has many practical applications, such as hardware verification (Clarke et al., 2003; Gupta et al., 2006), configuration management (Mancinelli et al., 2006; Tucker et al., 2007), or planning (Behnke et al., 2018). A realistic case study of SAT is illustrated in Example 3.

**Example 3** *Let us consider the problem of installing a new (software) package $P$ on a system where the installation is subject to dependency and conflict constraints. The goal is to determine which packages must be installed on the system such that the package $P$ is installed in the system, the dependencies of all the installed packages are satisfied, and there are no conflicts among the installed packages. This problem can be conveniently modeled as an SAT problem (Tucker et al., 2007). Formally, let $I$ be the set of packages involved in the installation, and let $x_i$ be a Boolean variable stating if the package $i$ in $I$ is installed. The constraints are encoded as follows: (1) $x_P$, it ensures that the target package $P$ is installed, (2) $x_A \to x_B$, it ensures that the package $A$ can be installed only if package $B$ is also installed (dependency between packages), (3) $\neg x_A \vee \neg x_B$, it ensures that packages $A$ and $B$ cannot be installed together (conflict between packages). Assuming that $C_1^d, \ldots, C_n^d$ are the dependency constraints, and that $C_1^c, \ldots, C_m^c$ are the conflict constraints, the Boolean formula to resolve corresponds to the logical conjunction of all the constraints, i.e.,*

$$\varphi = x_P \wedge C_1^d \wedge \cdots \wedge C_n^d \wedge C_1^c \wedge \cdots \wedge C_m^c.$$

*Many variants can be inferred from this formalization, such as integrating dependencies among more packages or finding the minimum set of packages that must be installed.*

A natural extension of SAT is the *maximum satisfiability problem* (MaxSAT), which aims to determine the maximum number of clauses, of a given Boolean formula in conjunctive normal form that can be evaluated to *true* by assigning truth values to the variables.

### 2.3.3 Constraint satisfaction and constraint 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*, i.e., CSPs that also have an objective function. The goal becomes 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 applied 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* $\mathbb{R}^{n\times n}$*, 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 read as*

$$\begin{aligned} &\min\ w_{x_n,x_1} + \sum_{i=1}^{n-1} w_{x_i,x_{i+1}} \\ &\text{subject to } \textsc{allDifferent}(x_1,\dots,x_n), \end{aligned}$$

*where* $\textsc{allDifferent}(X)$ *enforces that each variable from* $X$ *takes a different value (Régin, 1994), and the entries of the weight matrix* $w$ *are indexed using variables. This model forces 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 above, constrained problems can model arbitrary constraints and objective functions. This generality makes it possible to use general-purpose solving methods such as *local search* or *constraint programming*; see next section. In addition to their convenience on the modeling side, the high-level constraints generally referred to as *global constraints* are also useful on the solving side (Régin, 2004). They enable the design of efficient algorithms dedicated to pruning the search space. Leveraging the pruning ability of global constraints is a fundamental component of a constraint programming solver as explained below.

### 2.3.4 Solving CO problems

Major algorithmic frameworks—whose components and tasks have been recently considered through the GNN lens—will be discussed when necessary in the core of the survey. However, in this section, we briefly distinguish three algorithmic categories.[4]

**Exact methods** ILP models are generally solved to proven optimality (or proof of infeasibility) by variations of the *branch-and-bound algorithm* (Land and Doig, 1960; Lodi, 2010). Essentially, the algorithm is an iterative divide-and-conquer method that

1. solves LP relaxations (see Section 2.3.1),
2. improves them through valid inequalities (or *cutting planes*), and
3. guarantees to find an optimal solution through implicit enumeration performed by *branching*, see Figure 3.

As anticipated in Section 2.3.1, the quality of the LP relaxation plays a fundamental role in the effectiveness of the above scheme. Thus, step 2 above is particularly important, especially at the beginning of the search. The above scheme is called *branch and cut*. If integer programming techniques do not directly model the CO problem, then combinatorial versions of the branch-and-bound framework are devised, i.e., featuring relaxations different from the LP one, specifically associated with the structure of the CO problem at hand.

Specifically designed as an exact method for constrained satisfaction and optimization problems, *constraint programming* (CP) (Rossi et al., 2006) is a general framework proposing algorithmic solutions also within the divide-and-conquer scheme. 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 any more. A CP solver's efficiency relies heavily on its propagators' quality. Example 4 introduced the well-known allDifferent constraint. Its propagator (Régin, 1994) is based on maximum matching algorithms in a graph. Many other global constraints are available in the literature, such as Element constraint, allowing indexation with variables, or Circuit constraint, enforcing a set of variables to create a valid circuit. At the time of writing, the global constraints catalog reports more than 400 global constraints (Beldiceanu et al., 2005). The CP search commonly proceeds in a depth-first fashion, together with branch-and-bound. For each feasible solution found, 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 is guaranteed to be optimal.

Finally, although initially designed for solving decision problems, SAT solvers can also be used for combinatorial optimization. One way to do that is to specify objectives through soft constraints. The objective turns to satisfy as many soft constraints as possible in a solution,

4. We refer to Festa (2014) for additional details on the classification of algorithms for CO.

such as in the MaxSAT variant. Another option is to add a repertoire of commonly used objective functions in the solver and invoke the specialized optimization module when it corresponds to the objective function that must be optimized. *Satisfiability modulo theories* (SMT) solvers, a generalization of SAT, which can handle more complex formulas, generally support both options; see for instance Z3 solver (de Moura and Bjørner, 2008).

**Local search and metaheuristics** Local search (Potvin and Gendreau, 2018) is another algorithmic framework that is commonly used to solve general, large-scale CO problems. 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, a solution's *neighborhood* is the set of solutions obtained by modifying the solution $s$. In practice, local search algorithms are improved through *metaheuristic* concepts (Glover and Kochenberger, 2006), leading to algorithms like *simulated annealing* (Van Laarhoven and Aarts, 1987; Delahaye et al., 2019), *tabu search* (Glover and Laguna, 1998; Laguna, 2018), *genetic algorithms* (Kramer, 2017), *variable neighborhood search* (Mladenović and Hansen, 1997; Hansen et al., 2019), all of which are designed to help escape *local minima*.

**Approximation algorithms** The class of *approximation algorithms* (Vazirani, 2010) is designed to produce, typically in polynomial time, feasible solutions for CO problems. Unlike local search and metaheuristics, the value of those feasible solutions is guaranteed to be within a certain bound from the optimal one. Notable examples of approximation algorithms are *polynomial-time approximation schemes* (PTAS) that provide a solution that is within a factor $1+\epsilon$ (with $\epsilon > 0$ being an input for the algorithm) of being optimal (e.g., Arora (1996) for the TSP), or *fully polynomial-time approximation schemes* (FPTAS), where additional conditions on the running time of the algorithm are imposed (Ausiello et al., 2012).

### 2.4 Machine Learning

This section gives 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* highly relevant 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 that 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 (i.i.d.) with respect to a fixed but unknown distribution $\mathcal{D}$. Moreover, we assume that there exists a *target concept* $c\colon \mathcal{X} \to \mathcal{Y}$ that 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* $\widehat{R}(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(h(s_i), c(s_i))$, where $\ell\colon \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is the loss function. To avoid overfitting 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 logistic regression, support vector machines, and decision trees.

**Unsupervised learning** Unlike supervised learning, 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 has achieved its goal so far. The RL agent aims to maximize its cumulative reward 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 agent performs actions, yielding a sequence of states until reaching a terminal state, $s_\Theta$. 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 $\gamma$ 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.

**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

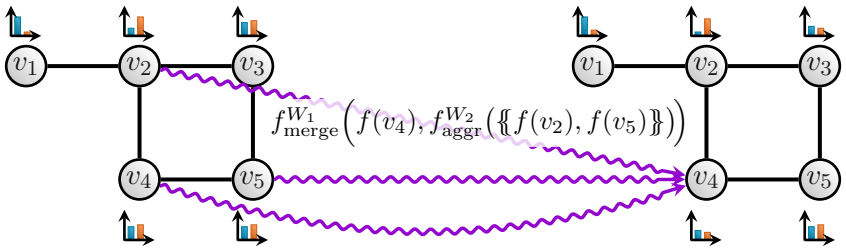



Figure 2: Illustration of the neighborhood aggregation step of a GNN around node $v_4$.

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 2 for an illustration. Formally, let $(G, l)$ be a labeled graph with initial node features $f^{(0)} \colon V(G) \to \mathbb{R}^{1\times d}$ 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 $l(u) = l(v)$, e.g., a one-hot encoding. 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 or, more formally, a composition of (permutation-invariant) functions. Each layer or function aggregates local neighborhood information, i.e., neighbors' features, within each node and then passes this aggregated information to the next layer.

For example, following (Morris et al., 2019), the aggregation of neighboring features can be realized via a simple component-wise sum, which naturally satisfies the requirement of a permutation-invariant function. That is, 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 $\sigma$ denotes a component-wise non-linear function, e.g., a sigmoid or a ReLU.[5]

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

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

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

5. For clarity of presentation, we omit biases.

parameters as $W_1$ and $W_2$, respectively. To adapt the parameters $W_1$ and $W_2$ of (1) and (2), they are optimized in an end-to-end fashion (usually via stochastic gradient descent methods) 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 or 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, possibly 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. The following practical scenarios motivate the need to quickly obtain high-quality feasible solutions, even without optimality or approximation guarantees.

**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 optimal value of a problem instance. For example, if the objective value represents financial cost, it might only care about how much profit is gained from switching from one resolution method to another. In this scenario then, heuristics are all that are needed. Moreover, there are many CO problems which are both practically intractable with an exact solver, and for which no proxy in the form of a good dual bound is easily computable as well. For these problems, a practitioner has no other guide to assess the suitability of a resolution method than to compare absolute objective values as well. For example, many vehicle routing problems admit strong MIP formulations with an exponential number of variables or constraints, similar to the TSP formulation in Example 2, see Toth and Vigo (2014). For such problems, simply computing the linear relaxation (typically using column or constraint generation (Dror et al., 1994)) is challenging, so a heuristic that consistently finds good solutions within a short user-defined time limit might 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 focused

on the primal side. A common use case is manually analyzing a good solution 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.[6] 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 develop a custom, standalone heuristic executed first, providing a warm start solution to the exact solver. This simple approach is widely used and addresses goals a) and b) simultaneously when the heuristic in question is effective for the problem of interest. This can also be done 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, categorizing them according to the learning paradigms of Section 2.4. In surveying the various works, we will touch on the following key aspects: the CO problem(s) tackled (e.g., TSP, SAT, MIP, graph coloring, etc.); the training approaches or loss functions used, with a focus on how hard constraints are satisfied; and GNN architecture choices. We do note that while, in some cases, the choice of architecture was intentional and well-justified, most works use one of the many interchangeable GNN architectures due to favorable empirical results or architectural novelty. This points to *principled architecture design for CO* as an impactful potential topic of future research, as discussed in Section 4.

#### 3.1.1 Supervised Learning

Assuming access to one or more optimal or near-optimal solutions to training instances of the CO problem of interest, a supervised learning approach is justified.

**One-shot solution prediction** 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, encoding the TSP instance, to a tour of a 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.

Prates et al. (2019) train a GNN in a supervised manner to predict the satisfiability of the decision version of the TSP. Small-scale instances of up to 105 cities are considered. This idea has been further extended by Lemos et al. (2019) for the decision version of the graph coloring problem. An important limitation of this last approach is that the model may generate infeasible solutions (violating some constraints). While this is expected, given that these two approaches tackle decision problems, it also motivates the need to design appropriate mechanisms for handling combinatorial constraints.

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

**Combining supervised models with search** 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 the approach of Vinyals et al. (2015), 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 subsequently analyzed by Joshi et al. (2022) using the TSP as a case study. It is shown that for this supervised learning approach to work well, combining it with the more expensive beam search is preferable. On the other hand, an RL policy can perform as well by acting greedily. This finding is not surprising as supervised learning with a single optimal solution per training instance is inherently limited when there are multiple optima (a common situation for CO problems).

Similarly, Li et al. (2018b) propose to solve combinatorial problems that are easy to reduce to the Maximum Independent Set (MIS) by a combination of supervised learning with a GNN model, and classical algorithms such as tree search. On a training set of instances where each problem is labeled with a precomputed optimal solution, the GNN is trained to output multiple continuous solution predictions, with a hindsight loss function (Guzman-Rivera et al., 2012) that considers the minimum (cross-entropy) loss value across the multiple predictions. As such, the training encourages the generation of diverse solutions. Then, 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. Finally, the authors report favorable results compared to some RL methods and non-learned heuristics. However, follow-up work by Böther et al. (2022) details, despite careful attempts, being incapable of reproducing the results, even reporting that using random weights in the GNN yields similar results as the trained weights. Thus, at this moment this approach should be considered at best inconclusive. Böther et al. (2022) however report more encouraging results with the unsupervised approach of Ahn et al. (2020), which we describe further below.

**Graph matching and related problems** Besides 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 misalignments due to the first stage's purely local nature, the authors propose a differentiable, iterative refinement strategy to reach a consensus of matched nodes.

Bai et al. (2020) introduces a GNN-based architecture, called GraphSIM, to solve the maximum common sub-graph and the graph edit distance problems. The idea is to generate vector representations for each node in the two graphs that must be compared, then compute similarity matrices based on the embeddings of every pair of nodes in the two graphs, and

finally use a standard convolutional neural network to obtain a similarity score between the two graphs. The training is carried out in an end-to-end fashion with supervision.

Nowak et al. (2018) train 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 to input a GNN.

**Guiding primal heuristics for MIP with supervised GNNs** Going beyond particular CO problems and towards more general frameworks, Ding et al. (2020) explore leveraging GNNs to heuristically solve MIPs by representing them as a tripartite graph consisting of variables, constraints, 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 use 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. Still, within the local branching framework, Liu et al. (2022) use a GNN to predict the initial size of the neighborhood to be explored by the algorithm, and they leverage reinforcement learning to train a policy that dynamically adapts the neighborhood size at the subsequent local branching iterations.

Nair et al. (2020) propose a neighborhood search heuristic for ILPs, called neural diving, that consists of a two-step procedure. 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. 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 kept variables. A binary classifier is trained to predict which variables should be thrown away at the second step. Notice how these supervised GNNs for MIP are all combined with some form of search that leverages the powerful MIP solver to guarantee that the hard linear constraints are satisfied.

**SAT and related problems** 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. It is stressed that the NeuroSAT system is still vastly less reliable than state-of-the-art SAT solvers. This is acknowledged by the authors (Selsam et al., 2019) who explain that the main motivation was to better understand the potential of neural networks capable of logical reasoning. As a suggestion for further works, they propose to use the NeuroSAT prediction inside a traditional SAT solver, instead of relying on end-to-end solving. This integration is discussed in more detail in Section 3.2. Another improvement has been proposed by Sun et al. (2020b). They conjectured that the limitations of NeuroSAT are mainly due to the difficulty of learning an all-purpose SAT solver. They evaluate this hypothesis by training another classifier, NeuroGIST, trained and tested only with SAT problems generated from differential cryptanalysis on the block cipher GIFT. Experimental

results show that the new model can perform better than the original NeuroSAT for this specific family of instances.

Abboud et al. (2020) learn an algorithm for estimating the model count (i.e., the number of satisfying assignments) of a Boolean formula expressed in a disjunctive normal form. To do so, they train with supervision a GNN on graphs representing formulae and output the parameters of a Gaussian distribution. The loss is the Kullback-Leibler (KL) divergence between the predicted and ground truth distributions.

#### 3.1.2 Unsupervised Learning

The approaches in this section use GNNs to produce solutions that simultaneously and directly optimize a CO problem's objective (if one exists) and minimize a measure of constraint violation (on average across a set of training instances). As such, no optimal solutions are required for the training instances, circumventing the limitations of the supervised paradigm. 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 adheres to the constraints. This idea has been also investigated by Amizadeh et al. (2018) for solving the circuit-SAT problem. Specifically, instances of such a problem are Boolean circuits and can be represented as a directed acyclic graph (DAG). The authors use directly the DAG structure as an input, as opposed to typical undirected representations of SAT problems such as in NeuroSAT. The training is carried out with no supervision and is based on an innovative loss function that, when minimized, pushes the model to produce an assignment that yields a higher satisfiability. Schuetz et al. (2022) follow a similar approach, applicable to CO problems that can be formulated as a quadratic unconstrained binary optimization problem (QUBO). Specifically, they express the QUBO as a graph by creating a node for each variable and connecting two such nodes if the corresponding variables interact in the QUBO objective. To make the QUBO objective differentiable, they resort to a relaxation approach, i.e., allowing the variables to take values from $[0, 1]$. A final integral solution is obtained via a simple rounding heuristic. Xu et al. (2020a) tackle the non-periodic 2D tiling problem, which consists of covering an arbitrary 2D shape using one or more types of tiles given as input. To do so, they propose a loss function containing three terms: (1) maximizing the tiling coverage of a region, (2) minimizing the tile overlap, and (3) avoiding holes in the shape. The loss is minimized by a self-supervised approach that does not need ground-truth tiling solutions for the training. The problem is modeled as a graph problem, and a new GNN architecture referred to as TilinGNN, is introduced. This method can efficiently cover many 2D shapes (moons, butterflies, turtles, etc.) involving more than 2 000 tiles.

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 a possible solution of the given graph problem[7], by minimizing a probabilistic penalty loss function. To produce an integral solution, they de-randomize the continuous values, using sequential decoding, showing that this solution obeys the given, problem-specific constraints with high probability. This result is noteworthy as most other works referenced in this survey lack theoretical guarantees of any kind.

7. Graph Partitioning and Maximum Clique are the examples considered in (Karalias and Loukas, 2020).

More recently, Duan et al. (2022) revisit the supervised NeuroSAT (Selsam et al., 2019) through the lens of "contrastive learning" (Chen et al., 2020), a popular unsupervised learning method. First, multiple views of every (unlabelled) SAT instance were generated through *label-preserving* augmentations: transformations that preserve the satisfiability of the instance. Then, the same GNN encoder of NeuroSAT was trained by maximizing the agreement between the representations of different views of the same instance *(positive pairs)* while minimizing the agreement between the representations of distinct instances *(negative pairs)*. They show that these representations can then be fine-tuned with much less labeled data than the fully-supervised method of NeuroSAT, to produce an equally accurate predictor of satisfiability. They argue that this method can be applied to problems beyond SAT to help reduce the sample complexity in various downstream tasks. Finally, Ozolins et al. (2022) show that for SAT problems evaluating a loss term after each term and minimizing their sum can lead to performance benefits over several baselines.

#### 3.1.3 Reinforcement Learning for Iterative Solution Construction

Common to the supervised and unsupervised approaches discussed thus far is that the GNN is used to produce values or scores for the decision variables being optimized over after which a feasible solution is directly read out or a search algorithm (beam search, MIP, etc.) is seeded based on the predicted variable scores. Alternatively, iterative construction algorithms are natural for many CO problems. This makes reinforcement learning an obvious candidate for automatically deriving data-driven heuristics.

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 the first to combine RL with deep neural networks in the CO setting. To overcome the need for near-optimal solutions in the approach of Vinyals et al. (2015), Bello et al. (2017) propose to train Ptr-Net models using policy gradient RL methods. However, this approach failed to address a fundamental modeling 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.

**GNNs enable RL-based policies for CO** Dai et al. (2017) leverage GNNs 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. 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. Dai 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, 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 depends on the particular combinatorial problem. However, the approach introduced by Dai et al. (2017) seems useful for various problems and admits many direct extensions and improvements, some of which we will survey next.

**The role of attention** Kool et al. (2019) tackle routing-type problems by training an encoder-decoder architecture using the REINFORCE RL algorithm (Sutton et al., 1999). This is based on Graph Attention Networks (Veličković et al., 2018), a well-known GNN architecture. Problems tackled by Kool et al. (2019) include the TSP, the 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. 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. Attention-based models for routing problems allow each city to learn the importance of other cities to its own representation. This inductive bias seems to be useful for combinatorial problems. Additionally, 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, etc. This allows for a unified learning approach to produce good heuristics for different optimization problems. Besides, François et al. (2019) show that the solutions obtained by Deudon et al. (2018); Kool et al. (2019); Joshi et al. (2019); Dai et al. (2017) can be efficiently used as the first solution of a local search procedure for solving the TSP.

**Handling hard constraints** The problems discussed thus far in this section have relatively easy-to-satisfy constraints. 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 constraints may be more difficult 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 learnable parameters generate feasible solutions, while others focus on minimizing costs. 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. (2021) 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: first, finding a feasible and complete solution, and second, finding the solution minimizing the objective function among the feasible solutions. The construction of a solution is stopped as soon as no action is available, which corresponds to an infeasible partial solution. Each complete solution obtained has then the guarantee to be feasible.

Ahn et al. (2020) propose to handle combinatorial constraints as follows. They introduce the notion of *deferred Markov Decision Process*, where at each iteration, the agent decides whether a decision should be carried directly or deferred to a subsequent iteration. This enables the agent to focus on the easiest decisions and to tackle the hardest decision only at the end when the set of candidate solutions is narrowed. Experiments are carried out on the maximum independent set problem, and competitive results are obtained for graphs having millions of vertices. However, the approach is limited to locally decomposable problems, where locally connected variables can decompose the feasibility constraint and the objective. The model is trained with an actor-critic reinforcement learning algorithm and is based on the GraphSAGE architecture (Hamilton et al., 2017).

**Other applications** For SAT problems, Yolcu and Póczos (2019) propose to encode SAT instances as an edge-labeled, bipartite graph and use 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 variable 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 various problems, e.g., 3-SAT, clique detection, and graph coloring. To overcome the sparse reward nature of SAT problems, they additionally employ curriculum learning (Bengio et al., 2009).

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). Liu et al. (2021) employ GNNs to learn chordal extensions in graphs. Specifically, they employ an on-policy imitation learning approach to imitate the minimum degree heuristic.

#### 3.1.4 Summary

In the above, we overviewed how GNNs can be leveraged to (quickly) find feasible solutions to CO problems in both the supervised and the unsupervised regimes. The overarching aim here is to find a feasible solution quickly rather than find optimal ones. In the supervised regime, approaches directly predict a feasible solution using a GNN or integrate the GNN's prediction into heuristics, e.g., some form of a tree or local search. In the former, guaranteeing that a predicted solution remains a challenge. Moreover, the costly curation of a large enough

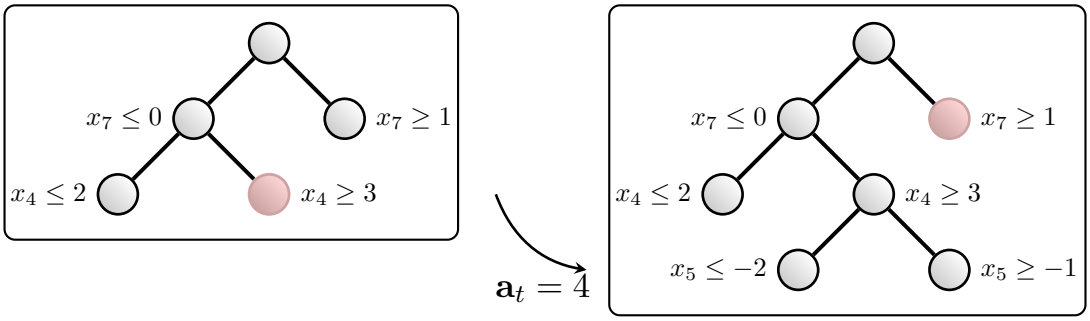



Figure 3: Variable selection in the branch-and-bound integer programming algorithm as a MDP.

training set usually hinders real-world applicability in both cases. Unsupervised approaches, e.g., based on reinforcement learning, address this problem. However, for many practically relevant CO problems, it remains challenging to handle constraints, and their performance is lacking behind heuristic approaches. Further, the high training time and computational needs of typical reinforcement learning approaches hinder real-world applicability. In the next section, we will focus on dual solutions. Here, the GNNs' predictions are integrated into exhaustive searches to guarantee that an optimal solution is found.

### 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. Determining such bounds 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 of validity of a bound), GNNs usually replace specific algorithms' components. In this context, GNNs are commonly combined with an exact solving process (see Section 2.3.4), such as *integer programming*, *logic solving*, *constraint programming*, or *branch-and-bound based on decision diagrams*.

#### 3.2.1 Integer programming

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. In the course of this algorithm, many decisions are repeatedly taken, whose influence is still poorly understood, and which have been described as the "dark" side of integer programming (Lodi, 2013). One of the most critical is the choice, at every iteration, of the variable whose range will be divided in two. As this choice significantly impacts the algorithm's execution time (essentially, the size of the branch-and-bound tree), 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 an MDP, as illustrated in Figure 3.

Preliminary work attempted the learning from fixed-dimensional representations of the problem, although without reaching the same level of performance as the best human-designed rules. In this regard, the usage of GNNs was a breakthrough and currently represents state-of-the-art. The first such GNN-based approach 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 those the model is trained on.

This work represented an important advance in that it was the first time that variable selection was shown to be improved by machine learning in a full-fledged integer programming solver, at least in some classes of instances. Nonetheless, it quickly became evident that there were many aspects of the approach that could be improved, which led to a succession of works that tried to further improve on the method. One concern that was quickly brought up was that their approach requires the usage of a GPU at inference time, which might be impractical. 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. Another concern that was brought up is that the resulting policy is specialized to a type of problem rather than being a good universal policy. Nair et al. (2020) address the latter, first by proposing a GPU-friendly parallel linear programming solver using the alternating direction method of multipliers (ADMM) approach and second, by showing that it could be used to scale the training to much larger datasets. In particular, they showed that they could learn a policy that performed well on MIPLIB (Gleixner et al., 2020), the gold standard benchmark for solvers, although at a high computational cost. Nonetheless, it is a remarkable achievement that suggests that the GNN approach can generalize over very heterogeneous datasets, perhaps all MIPs, simply by scaling up.

Finally, an alternative criticism is that the performance of the imitation learning policy will never exceed that of the expert, strong branching, which is known to perform badly in some families, such as multiple knapsack problems. This suggests that reinforcement learning methods could be a better long-term solution for training graph neural networks to select variables in integer programming solvers. The preliminary work of Sun et al. (2020a) hints in this direction by showing that evolution strategies can offer improvements over small homogeneous datasets. However, one severe limitation on using RL for branching is that, in the presence of large branch-and-bound trees, it is very difficult to perform the so-called *credit assignment*, i.e., selecting which RL actions should be credited for a specific outcome. Two recent works (Etheve et al., 2020; Scavuzzo et al., 2022) show that the structure itself of the branch-and-bound tree can be leveraged in the attempt to simplify the credit assignment, leading to a faster convergence of RL algorithms.

So far, most of the work in this area has focused on variable selection. Nonetheless, many other components of integer programming solvers seem just as amenable to machine learning approaches. Other aspects have started to be approached with GNN methods, such as column selection for column generation (Morabit et al., 2021; Chi et al., 2022), node selection (Labassi et al., 2022) or cutting plane selection. Cutting planes are additional constraints that simplify the search process yet are guaranteed not to remove the optimal

solution and are continuously generated throughout the solving process. As one does not want to include all generated cutting planes for fear of rendering the computational burden unnecessarily heavy, only the most "useful" cuts are to be added, something which is difficult to estimate. Paulus et al. (2022) use GNNs over a tripartite graph representation inspired by the bipartite representation of Gasse et al. (2019) and show improvements over standard cut selection procedures in a variety of benchmarks. Also of note is the work of Khalil et al. (2022), where a GNN is used to predict the most likely value of variables in optimal solutions of binary integer linear programs. This, in turn, is used in their work to guide the selection of the next node to explore, as well as warm-starting the solver by rounding the predicted values combined with solver-provided "solution repair" heuristics.

Naturally, these works can be extended in many directions, and many other aspects of integer programming solvers seem ripe for improvement by machine learning methods, especially using GNN models. As described, many works already outperform the state-of-the-art designed by humans, and it seems likely that solvers will integrate, over time, many such learned rules in their systems. One could, in fact, imagine an online training scenario where such components would continuously learn, and solvers would improve on the type of problem of interest to the practitioner each time it solves a problem.

Finally, a closely related problem to integer linear programming is neural network verification, where a branch-and-bound algorithm is also used as the backbone solution method. In fact, the properties of a neural network are often verified by solving a mixed-integer optimization problem. Lu and Kumar (2020) 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. Although this area has received comparatively less attention than general integer programming, their proximity suggests many opportunities for translating advances in that field to neural network verification and, maybe, vice versa.

#### 3.2.2 Logic solving

In logic solving, such as for Boolean Satisfiability, Satisfiability Modulo Theories, and Quantified Boolean formulae, replacing human-designed decision rules with trained GNN models has also brought improvements, although the wider variety of algorithms and competing approaches make the impact less recognizable than in integer programming so far. Nonetheless, the analogies are strong enough for advances in one field to extend to the other. In particular, the role of branch-and-bound as the core algorithm is instead taken by Conflict-Driven Clause Learning (CDCL), a tree search algorithm that constructs a solution through a combination of backtracking and learning new clauses from conflicts (Marques-Silva et al., 2021). Analogously to branch-and-bound in integer programming, in this algorithm, one must repeatedly "branch", which in this case involves picking both an unassigned variable and a polarity (value) to assign to this variable.

In this context, some authors have proposed representing logical formulae as graphs and using a GNN to select the best next variable, the analog of a branching step. Namely, Lederman et al. (2020) model quantified boolean formulae as bipartite graphs and teach a GNN to branch using REINFORCE. Although the reinforcement learning algorithm is

very simple, they achieve substantial improvements in several formulae solved within a given time limit compared to VSIDS, the standard branching heuristic. Two other works apply similar ideas to related problems. Kurin et al. (2020) model propositional Boolean formulae as bipartite graphs and train a GNN to branch with $Q$-learning. Although the problem is different, they similarly show that the learned heuristic can improve on VSIDS, namely in the number of iterations needed to solve a given problem. Moreover, Vaezipoor et al. (2021) represent propositional Boolean formulae as bipartite graphs and train a GNN to branch with evolution strategies, but on a Davis–Putnam–Logemann–Loveland solver for #SAT, the counting analog of SAT. They show that this improves solving time compared to SharpSAT, a state-of-the-art exact method.

It is interesting to note here that these works all rely on reinforcement learning, in contrast with integer programming, where this approach, so far, is much less successful compared to imitation learning. One possible explanation is that the boolean nature of the variables and of the formulae makes the problem combinatorially less complex; another is those good experts are less obvious for these problems. Nonetheless, integer programming experience suggests that imitation learning methods, despite their drawbacks, could have a strong impact.

Finally, it is worth mentioning an approach that resembles more closely to imitation learning. We already mentioned in Section 3.1 the work of Selsam et al. (2019) that trains a GNN "NeuroSAT" architecture to predict the satisfiability of SAT formulae based on bipartite variable-clause graph representations. In follow-up work, Selsam and Bjørner (2019) train the same architecture to predict the probability of a variable being in an unfeasible core based on pre-solved formulae. This is used to inform variable branching decisions inside the MiniSat, Glucode and Z3 SAT solvers through the assumption 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. Although the approach is SAT specific, it suggests that an alternative approach to variable selection can be achieved in a more indirect way by training models to predict optimal solutions and using those predictions to guide solving, as also partially done by Khalil et al. (2022).

### 3.2.3 Constraint programming

The approach to variable selection found in integer programming and logic solving can also be extended to constraint programming (CP). In CP, standard algorithms, such as branch-and-bound, iterative limited discrepancy search, and restart-based search, are all variants of backtracking search algorithms where one must repeatedly select variables and corresponding value assignments. Value selection, in particular, significantly impacts the quality of the search.

In the case of constraint satisfaction programs that can be formulated as MDPs on graph states, such as the TSP with time windows, Cappart et al. (2021) 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 is been further extended by Chalumeau et al. (2021), who propose a new CP solver that natively handles a learning component. To do so, they

represent a CSP as a simple and undirected tripartite graph on which nodes represent each variable, possible value, and constraint. 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. This representation has the benefit of being generic to any combinatorial problem and has been reused by Song et al. (2022) for learning variable ordering heuristics. However, a current challenge is the size of the generated graph, which can be prohibitive, making the training more tedious. Similarly, van Driel et al. (2021) leverage GNNs to initialize a variable-selection heuristic inside a hybrid CP-SAT solver.

#### 3.2.4 Decision diagrams

A similar situation holds with decision diagrams (DD) which are graphs that can be used to encode the feasible space of discrete problems to obtain dual bounds in CO problems (Bergman et al., 2016). In many 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. This mechanism is particularly interesting as the bounds obtained are flexible, meaning that their quality depends on algorithmic choices made during the diagram's construction, unlike, e.g.,, the linear relaxation bound found in integer programming. Unfortunately, finding the algorithmic choices yielding the best bounds is often NP-hard. For instance, it is the case for determining the best variable ordering to build the diagram. Cappart et al. (2019) tackle this problem by training a GNN with 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 and the problem representation as a graph are similar to the one proposed by Dai et al. (2017). This idea is leveraged by Parjadis et al. (2021) and Cappart et al. (2022). They integrate the bounds obtained through this mechanism into a full-fledged branch-and-bound algorithm. The experimental results show that the learned bounds allow reducing the solving time for the maximum independent set problem compared to methods based on non-learned bounds or on linear relaxations.

When designing such an approach, an important consideration is the increased computational power required to obtain the bounds through a deep architecture. As the model must be called many times inside the branch-and-bound tree, it is important that the inference is carried out efficiently. This issue is further discussed in Section 4. This approach also suffers from some limitations, such as the need to be able to encode the problem into a decision diagram, making this strategy not suited for solving any kind of CO problem.

#### 3.2.5 Summary

In the above, we overviewed the application of GNNs to prove that a given solution is optimal. Here, in the area of integer programming, GNNs show promising performance to replace common heuristics within solvers. For example, the seminal work of Gasse et al. (2019) showed that GNNs can imitate costly variable selection heuristics, often beating the solver's default heuristic and initiating many follow-up works. Especially, Nair et al. (2020) proposed several refinements that showed promising performance on the notoriously hard MIP instances. However, all such approaches suffer from the need to compute a large

enough training dataset. However, recently, several works, e.g., Sun et al. (2020a); Scavuzzo et al. (2022), tried to address this limitation via reinforcement learning-based approaches. Interestingly, in logic solving, e.g., Boolean Satisfiability, several reinforcement learning-based approaches have been successfully applied, e.g., to learn branching heuristics. However, similar approaches have been tried for CP and decision diagrams, with mixed success.

### 3.3 Algorithmic Reasoning

We now turn our attention to *neural algorithmic reasoning* (Veličković and Blundell, 2021), a paradigm aiming to build neural networks that align with invariants and properties of classical algorithms (Cormen et al., 2009). While initially conceived as a way to probe, theoretically and empirically, the extent to which neural network architectures can solve classical reasoning tasks, the area holds the potential for attacking combinatorial problems over natural, noisy inputs (Deac et al., 2021). Further, it offers a framework for building neural networks in the presence of an algorithmic prior, which demonstrated significant returns for applications in pure mathematics (Davies et al., 2021; Blundell et al., 2022).

Interest in such a direction persisted over many years, with several works investigating the construction of general-purpose neural computers, e.g., the neural Turing machine (Graves et al., 2014), the differentiable neural computer (Graves et al., 2016), and its variants (Csordas and Schmidhuber, 2019). While such architectures have all the hallmarks of general computation, they introduce several components at once, making them often challenging to optimize. 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, focusing mainly 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).

In recent times, the work in neural algorithmic reasoning has consolidated towards using graph-structured models—that is, GNNs—at the core and moving beyond primitive algorithms towards combinatorial algorithms of super-linear complexity. The primary motivation for using GNNs comes from the framework of algorithmic alignment (Xu et al., 2020b; Dudzik and Veličković, 2022). Through this framework, an argument is made that GNNs are potentially capable of executing polynomial-time dynamic programming algorithms (Bellman, 1966), a paradigm from which many polynomial-time algorithms of interest can be constructed. We will investigate this framework in more detail throughout this section.

Studying supervised algorithmic execution tasks brings up another important issue of training neural networks. Namely, (G)NNs 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 in *extrapolation*, i.e., when evaluated out of distribution. For example, 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. Extrapolation 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 solved, it could be helpful 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 context. 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. Furthermore, even if a problem is in P, a GNN needs to have sufficient depth and width to be able to solve it (Loukas, 2020). Hence, when we say a GNN extrapolates on a hard CO task, we will not imply that the GNN will produce optimal solutions for arbitrarily large inputs. Instead, we will require the GNN to produce solutions that align with an appropriate polynomial-time heuristic, see Section 3.3.1. Similarly, extrapolation to the entire set of plausible inputs may not even be relevant. We may, instead, only care about extrapolation on a diverse[8] set of inputs that are relevant for the real-world task being solved. Training algorithmic neural networks capable of extrapolating only over those instances would still be valuable to an SAT solver portfolio, even if their general extrapolation power may be substantially limited beyond those instances.

Building neural networks that extrapolate also has a natural corollary, i.e., being able to generalize across completely different kinds of problems. For example, in the work of (Li et al., 2018c), a model is trained on the 3-SAT problem (by reduction to the maximal independent set problem), and it is still able to competitively solve related tasks, like finding maximum cliques or vertex covers. A better understanding of algorithmic alignment could also help us foretell which classes of related problems could benefit most from such a transfer. An example is work on learning local heuristics from Yolcu and Póczos (2019), where training on instances of one decision problem only transfers well to problems that are *similar* to it, such as vertex covering and finding dominating sets.

### 3.3.1 Algorithmic alignment

The concept of *algorithmic alignment* introduced by Xu et al. (2020b) 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 network components and control flow to line up well with the underlying algorithm to be learned from data. This section will use Figure 4 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. Those works concretize the theoretical findings of Xu et al. (2020b).

The work of Veličković et al. (2020) on the neural execution of graph algorithms is among the first such papers. It focuses entirely on the practical implications of learning to execute abstract algorithmic tasks. Accordingly, it suggests several general-purpose modifications to GNNs to make them extrapolate better on combinatorial reasoning tasks.

8. For example, it is well known that existing powerful SAT solvers can struggle over specific instances of logical formulae such as cryptography (Ganesh and Vardi, 2020).

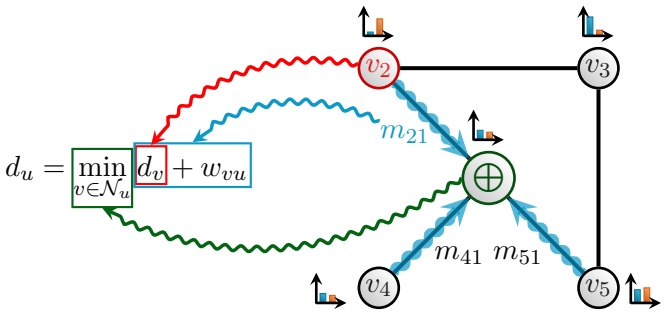



Figure 4: Illustration of algorithmic alignment, in the case of the Bellman-Ford shortest path-finding algorithm (Bellman, 1958). It computes distance estimates for every node, $d_u$, and is shown on the left. Specifically, a GNN aligns well with this dynamic programming update. Node features align with 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 neighbors (**green**).

**Using the *encode-process-decode* paradigm (Hamrick et al., 2018)** Inputs, $x$ in $\mathcal{X}$, are encoded into latents, $z$ in $\mathcal{Z}$ (e.g., $\mathcal{Z} = \mathbb{R}^d$ for $d > 0$), 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. Such a factorized computational model allows for easier modeling of the algorithm's *intermediate state* (by decoding at different points in executing $P$). It also allows for sharing the processor across several tasks if we assume any relevant computation can be reused across them.

**Favoring the (component-wise) max aggregation function** This aligns well with the fact combinatorial algorithms often 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 for larger inputs and neighborhoods. Such findings have been independently verified empirically (Joshi et al., 2022; Richter and Wattenhofer, 2020; Corso et al., 2020) and contradict the common advice to use sum aggregation (Xu et al., 2019). Algorithmically aligned aggregation has also recently been generalised beyond tasks requiring "pure" maximisation, to tasks leveraging arbitrary *commutative monoids* by Ong and Veličković (2022).

**Leveraging strong supervision with teacher forcing (Williams and Zipser, 1989)** If, at training time, we have access to execution traces from the ground truth algorithm, which illustrates how input data is manipulated[9] 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

9. In this sense, for sequences of unique inputs all correct sorting algorithms have the same input-output pairs, but potentially different sequences of intermediate states.

more closely follow the ground-truth algorithm's iterations (Hussein et al., 2017). This provides a mechanism for encoding and aligning with algorithmic pre- and post-conditions, e.g., after $k$ iterations of a shortest-path algorithm such as Bellman-Ford (Bellman, 1958), the shortest paths that use up to $k$ hops from the source node can be computed. Strong supervision works well even without access to execution traces, as is demonstrated by the RRN model of Palm et al. (2018). 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 unnecessary. Many efficient combinatorial algorithms are efficient precisely because they focus on only a small amount of nodes 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, as done in Yan et al. (2020)) can therefore be impactful and at times more important than the choice of processor.[10]

**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 (Xhonneux et al., 2021; Numeroso et al., 2023) or using the output of simpler algorithms as "latent input" for more complex ones, significantly improving empirical performance on the complex task. Recently, it was shown that many diverse algorithms can be meaningfully learnt by the same processor network in this way, performing at single-task expert level on average (Ibarz et al., 2022).

While initially applied only to path-finding and spanning-tree algorithms, the prescriptions listed above have been applied for heuristically solving bipartite matching (Georgiev and Lió, 2020), mazes (Schwarzschild et al., 2021; Bansal et al., 2022), max-flow/min-cut (Awasthi et al., 2022; Numeroso et al., 2023), model-based planning (Deac et al., 2020, 2021; He et al., 2022), network configuration protocols (Beurer-Kellner et al., 2022) and TSP (Joshi et al., 2022). It is worth noting that, concurrently, significant strides have been made on using GNNs for physics simulations (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2020). These proposals came up with a largely equivalent set of prescriptions to the ones discussed above.

Several works have expanded on these prescriptions even further, yielding stronger classes of GNN executors. PrediNets (Shanahan et al., 2020) are capable of forming representations of propositions in the context of propositional logic, and ACER (Georgiev et al., 2022) further demonstrates that correct, interpretable propositional formulae can be extracted from trained PrediNets for certain tasks, such as computing minimum spanning trees. IterGNNs (Tang et al., 2020) align well with a broad class of algorithms that repeatedly execute some computation until a certain stopping criterion is satisfied. The design of IterGNNs allows for adaptively learning the stopping criterion without requiring an explicit

10. For example, Veličković et al. (2020) show empirically that, for learning minimum spanning tree algorithms, LSTM processors with the masking inductive bias perform significantly better out of distribution than GNN processors without it.

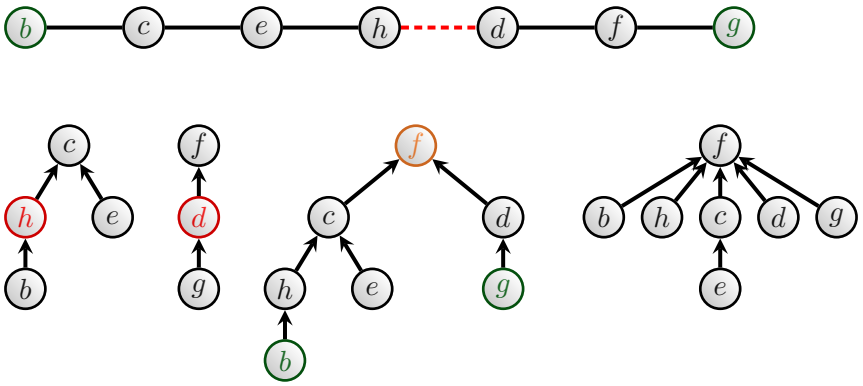



Figure 5: The utility of dynamically choosing the graph to reason over for incremental connectivity. It is easy to construct an example *path graph* (*top*), wherein deciding whether one vertex is reachable from another requires linearly many GNN iterations. This can be ameliorated by reasoning over different links—namely, ones of the disjoint set union (DSU) data structure (Galler and Fisher, 1964) that represent 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 $(b, g)$ is 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.

network to predict when to terminate. 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.[11]

Recently, significant interest was also dedicated to inferring the graph over which the (G)NN should operate when solving a combinatorial task. Neural shuffle-exchange networks (Freivalds et al., 2019; Draguns et al., 2021) directly fix connectivity patterns between nodes based on results from routing theory (such as Beneš networks (Beneš et al., 1965)), aligning them with $O(n \log n)$ sequence processing algorithms. Lastly, pointer graph networks (PGNs) (Veličković et al., 2020) take a more pragmatic view of this issue. Rather than trying to fix a graph used by the processor GNN upfront (which may not even be given in many problems of interest), PGNs explicitly predict a graph to be used by the processor, enforcing it to match data structures' behavior.

As a motivating example, PGNs tackle the *incremental connectivity* task (Figure 5). Here, the model needs to answer queries of the form: given two nodes in an undirected graph, $u$ and $v$, does a path exist between them (i.e., is $v$ reachable from $u$?)? The graph can be modified between two queries by adding one edge at a time. It is easy to construct a worst-case "path graph" for which answering such queries would require $\Omega(n)$ 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 querying reachability in $O(\alpha(n))$ amortized complexity (Tarjan, 1975), where $\alpha$ is the inverse

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

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.

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.[12] 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 nodes is performed, storing their new latents. This effectively endows PMP with an episodic memory (Pritzel et al., 2017) of its past computations. Further, it has the potential to overcome more general problematic aspects in learning GNNs, such as over-smoothing (see also Section 4), beyond the realm of what is possible with simple approaches like skip connections: since latent features are never overwritten, there exist no means for them to ever get smoothed out in the future.

#### 3.3.2 Perspectives and outlooks

According to the previous discussion, algorithmically-aligned GNNs have already been explored in the context of dynamic programming, iterative computation,[13] as well as algorithms backed by data structures. While novel architectures continue to be developed, interesting theoretical results also elucidate what makes an architecture extrapolate well on algorithmic execution tasks. We summarize and refer to some of these theoretical results here and highlight two emerging outlooks on algorithmic GNNs.

Recent theoretical results have provided a unifying explanation for why algorithmically-inspired prescriptions benefit extrapolating both in algorithmic and physics-based tasks (Xu et al., 2021). Specifically, the authors make a useful geometric argument: ReLU-backed MLPs, being piece-wise linear functions, always tend to extrapolate linearly outside of the support of the training set. Hence, if we can design architecture components or task featurization 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 (component-wise) max aggregation performs well for shortest path-finding. The Bellman-Ford dynamic programming rule (e.g., as in Figure 4)

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

is an edge-wise linear function followed by a minimization. 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. Recent research has taken

12. In reality, each node in typical GNN stores a $d$-dimensional real vector. However, for most GNN architectures used in practice today, $d \leq 2,048$, so $O(d)$ can be treated as a constant for this analysis.
13. Recent work (Yang et al., 2021) has also demonstrated that GNNs can be made to align with iterative optimization algorithms, such as proximal gradient descent and iterative reweighted least squares.

this insight further, demonstrating that algorithmic extrapolation may necessitate either maintaining a *causal model* of the distribution shift (Bevilacqua et al., 2021), or carefully crafted self-supervised learning objectives (Yehudai et al., 2021).

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 is demonstrated to be possible. That is, neural execution engines (NEEs) (Yan et al., 2020) are capable of empirically 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 binary representations of inputs, and conditionally mask the computation; that is, at every step, a prediction is made on which nodes are relevant, and then a GNN is executed only over the relevant nodes. 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. For example, Dijkstra's algorithm (Dijkstra, 1959) and Prim's algorithm (Prim, 1957) have an identical implementation backbone—the main difference is in the key function used for a priority queue. This allows Yan et al. (2020) to directly re-use the components learned in the context of one algorithm when learning the other.

Lastly, an important concurrent research direction that shares many insights with algorithmic reasoning is *knowledge graph reasoning*. Briefly put, this area is concerned with expanding the body of knowledge in a (usually closed-domain) knowledge base, typically by inferring additional links or answering logical queries (Hamilton et al., 2018; Ren et al., 2019). Answering logical queries can often be helped by path-finding primitives, which was recently embodied in the NBFNet model (Zhu et al., 2021). NBFNet is a knowledge graph reasoning system designed to align algorithmically to generalized versions of the previously discussed Bellman-Ford algorithm.

#### 3.3.3 Reasoning on natural inputs

Until now, we have focused on methodologies allowing 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.[14] However, an obvious question arises: if all we are doing is imitating a classical algorithm, why not just apply the algorithm?

Many potential applications of algorithmic reasoning may provide answers to this question in principle.[15] However, one particularly appealing direction for CO has already emerged—

14. In principle, any function could be a correct extrapolant if the underlying target unknown known.
15. 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. 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) detect data-driven sorting procedures that seem to improve on quicksort, and Veličković et al. (2020) indicate, on small examples, that they can generalize the operations of the disjoint-set union data structure in a GPU-friendly way.

algorithmic learning executors allow 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 constraints. 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 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 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.

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 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 affect the applications of classical CO algorithms. Satisfying 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 be only partially observable, which can often render the algorithm completely inapplicable.

The recent "Amazon Last Mile Routing Research Challenge" (2021) (Winkenbach et al., 2021) serves as early evidence of this recognition in a high-stakes setting.[16] The challenge is motivated by the fact that

16. `https://routingchallenge.mit.edu/`

> *"there remains an important gap between theoretical route planning and real-life route execution that most optimization-based approaches cannot bridge. This gap relates to the fact that in real-life operations, the quality of a route is not exclusively defined by its theoretical length, duration, or cost but by a multitude of factors that affect the extent to which drivers can effectively, safely, and conveniently execute the planned route under real-life conditions."*

These factors involve additional contextual features and tacit knowledge (e.g., the driver's familiarity with certain routes or their observations of traffic) that would typically be dismissed if one models the routing problem using path lengths as the only data to optimize over.[17]

On the surface, these issues appear to be fertile ground for neural networks. Their capabilities, as a replacement for human feature engineering and a powerful raw data processor, are highly suggestive of their potential applicability. However, this can run into several obstacles. The first one concerns the learnability of such systems via gradient descent, as 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. However, promising ways to tackle the issue of gradient estimation have already emerged[18] in the literature (Knöbelreiter et al., 2017; Wang et al., 2019; Vlastelica et al., 2020; Mandi and Guns, 2020; Niepert et al., 2021).

A more fundamental 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,[19] if any component of the neural representation ends up poorly predicted, other components are still able to step in and compensate. This partly enabled neural networks' emergence as a flexible tool for raw data processing. If there is insufficient data to learn how to compress it into inputs expected by the algorithm 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 power comes from using the aforementioned encode-process-decode framework. Assume we have trained a GNN executor to perform a target algorithm on many abstract algorithmic 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.

17. It could be possible to encode this knowledge as classical CO constraints. However, the exact manner in which this can be done, especially if we want it to generalize across different drivers, is unlikely to be simple.
18. 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.
19. 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 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.

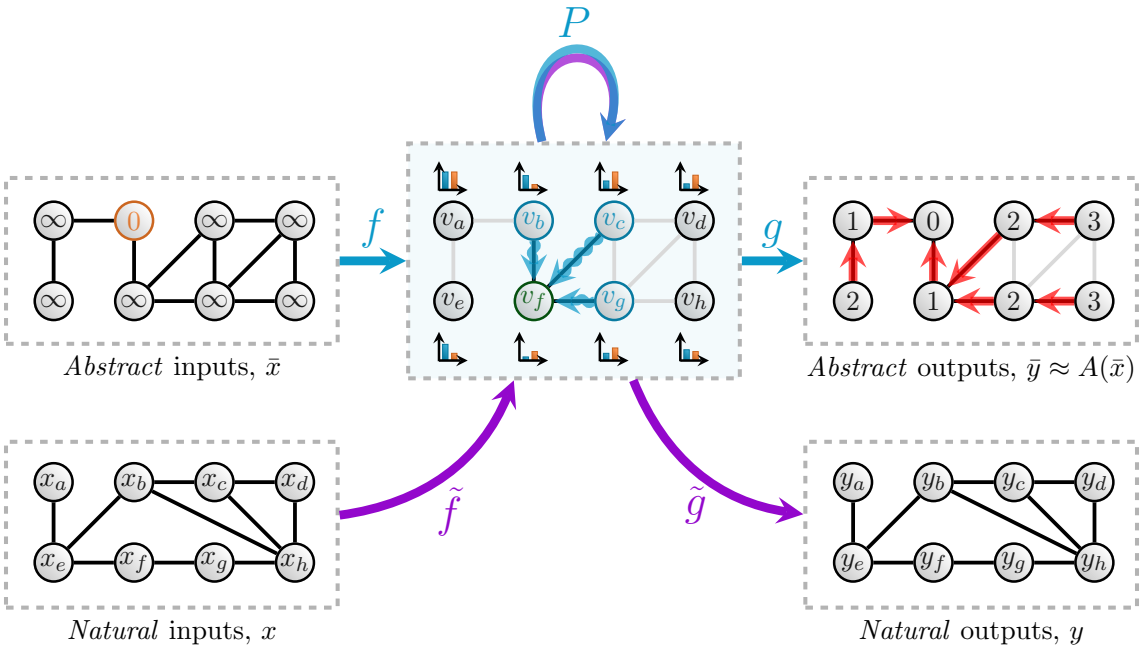



Figure 6: The proposed algorithmic reasoning blueprint. First, an algorithmic reasoner is trained in the encode-process-decode fashion, learning a function $g(P(f(\bar{x}))) \approx A(\bar{x})$, for a target combinatorial algorithm $A$; in this case, $A$ is breadth-first search. Once trained, the processor 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.

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 and may be more data efficient.

Such a processor thus seems to be a perfect component in a neural end-to-end pipeline that goes straight from raw inputs to general outputs. The general procedure for applying an algorithm $A$ (that admits abstract inputs $\bar{x}$) to raw inputs $x$ is as follows (see Figure 6):

1. Learn an algorithmic reasoner for $A$, on generated abstract 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.[20] 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 $P$.

20. 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.

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[21]—to reasoning over natural inputs. The raw encoder function $\tilde{f}$ learns how to map raw inputs onto the algorithmic input space for $P$—a task analogous to the human feature engineer—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 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[22] VI aligns with convolution. This yields the Value Iteration Network architecture (Tamar et al., 2016, VIN) that is a CNN-based agent, where certain convolutions share parameters in a manner that aligns with value iteration. While this construction can outperform baseline CNN agents in terms of generalization, and its extensions to graph-based environments using GNNs have been made (Niu et al., 2018), the above strong constraints on the MDP remained.

In the XLVIN architecture, Deac et al. (2021) surpass these limitations by following precisely the algorithmic reasoning blueprint above. They pre-train an algorithmic executor for VI on several randomly generated, known MDPs, then deploy it over a local neighborhood of the current state, derived using self-supervised learning. CNAP (He et al., 2022) extends these results even further by allowing planning over a continuous space of actions (enabling applications in robotics).

The representations produced by this VI executor substantially improve a corresponding model-free RL baseline, especially regarding data efficiency. Additionally, the model performs strongly in the low-data regime against ATreeC (Farquhar et al., 2018) that resorts 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. As shown further by Deac et al. (2021), even over challenging RL environments such as Atari, neurally learned algorithmic cores prove to be a viable way of applying classical combinatorial algorithms to natural inputs in a way that can 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 CO applications.

21. While presenting an earlier version of our work, Max Welling raised a very important point: if we aim 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 difficult 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 face 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.
22. Note that this does not ameliorate the requirements listed above. Assuming an environment is a grid-world places strong assumptions on the underlying MDP.

#### 3.3.4 Summary

In the present subsection, we have focused on constructing GNN architectures that are capable of executing individual steps of algorithms in a high-dimensional latent space. Neural algorithmic reasoning draws on a significant body of work on learning-to-execute, encompassing both a rich history of prior work on differentiable neural computers (Graves et al., 2014, 2016) and modern developments in modular, algorithmically-aligned (Xu et al., 2020b, 2021) architectures (Veličković et al., 2020). We showed the potential of neural algorithmic reasoning for going into the realm of general-purpose machine learning tasks with *algorithmic priors* (Deac et al., 2021; He et al., 2022; Veličković et al., 2022b; Numeroso et al., 2023). With the recent progress in learning generalist algorithmic reasoning agents (Ibarz et al., 2022), such modules could become viable parts of the next generation of artificially-intelligent agents.

Many challenges remain, of course—as evidenced by the large body of work investigating algorithmic alignment only recently, we are yet to understand the fundamentals of what does it mean to create truly robust neural executors (Bevilacqua et al., 2021; Yehudai et al., 2021; Dudzik and Veličković, 2022). Beyond this, significant work is needed to properly endow these models with a notion of *thinking time*, *memory*, and similar constructs.

## 4. Limitations and Research Directions

In the following, we give an overview of works that quantify the limitations of GNNs and the 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 explored the limitations of GNNs (Xu et al., 2019; 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 being unable 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., 2021; Abboud et al., 2021) indicate that randomly initialized node features can help boost the expressivity of GNNs, although the impact of such an approach on generalization remains unclear.

**Generalization of GNNs** To successfully deploy supervised machine learning models for CO, understanding generalization (out-of-training-set performance) is crucial. For example, 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.

**Other limitations of GNNs** Besides understanding generalization and expressivity in the context of combinatorial optimization, other GNN issues might also hinder their success in the realm of CO. For example, GNNs' node features, under specific assumptions, become indistinguishable, a phenomenon referred to as *over-smoothing* (Li et al., 2018a). Moreover, for GNNs, the *bottleneck problem* (over-squashing) refers to the observation that large neighborhoods cannot be accurately represented (Alon and Yahav, 2020). These problems prevent both methods from capturing global or long-range information, which is important for several CO problems, e.g., shortest-path applications. While there are some empirical studies (Dwivedi et al., 2020; You et al., 2020) investigating how different architectural design choices, e.g., skip connections or layer norm, circumvent the above-mentioned problems, a theoretical understanding is still lacking.

**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) proves that some GNNs can be too small to compute some properties of graphs, such as finding their diameter or a minimum spanning tree, and gives 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. If the inference is time-consuming, the overall wall-clock time of a solver or heuristic may not be reduced, even if the total number of iterations is smaller. A typical example is repeated decision-making within a CO solver, e.g., branching or bounding. 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. 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. Another option is to limit the number of times the GNN is invoked inside the solver, for instance, when the decisions are the most critical (Cappart et al., 2022).

**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 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 CO problems are rarely worst-case. Solving NP-hard problems for particular (real-world) distribution can be much easier than constructed, hard worst-case instances. Many important CO problems can be solved in polynomial time when considering some constraints on the input graphs. For example, developing a polynomial time algorithm is often easy if the considered graphs are trees or tree-like. Furthermore, it is known that the maximum independent set and the maximum clique problems can be solved in polynomial time for perfect graphs even though the problems are NP-hard.

**Achieving state-of-the-art performances** While progress has been made in the past few years with GNN approaches for CO, the state-of-the-art is still mainly dominated by specialized solvers and heuristics. As a concrete example, Angelini and Ricci-Tersenghi (2022) highlighted that a simple, linear-time, greedy heuristic outperforms modern GNNs (Schuetz et al., 2022). See also the discussion in Section 3.1.2 for the maximum independent set problem. Another example, emphasized by Yolcu and Póczos (2019), is that WalkSAT, a traditional local search algorithm introduced in the nineties (Selman et al., 1994), matches the performance of a GNN trained to perform a stochastic local search through reinforcement learning. Yao et al. (2019); Karalias and Loukas (2020) have made similar observations. Finally, most of the problems currently tackled by GNN approaches in the literature are generally much easier than what can be solved by traditional approaches. For example, in the case of the traveling salesperson problem, Joshi et al. (2022) highlighted that GNNs can obtain promising results for trivially small instances up to a few hundred cities. However, the well-known Lin-Kernighan-Helsgaun algorithm (Helsgaun, 2000; Taillard and Helsgaun, 2019) finds solutions close to optimality to instances of $10^7$ cities (with a duality gap of 0.588%, according to the Held-Karp lower bound[23]). That being said, there is evidence that good integration has already reached viable results. For example, this is the case of the GNN use for branching (Gasse et al., 2019) as well as for cut selection (Paulus et al., 2022), i.e., ML-augmented MIP tasks. However, in the same spirit, competitive results have been obtained in the enhancement of primal heuristics for MIP (Liu et al., 2022), or in physics simulations (Sanchez-Gonzalez et al., 2020; Pfaff et al., 2020).

### 4.2 Proposed New Directions

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

**Understanding when GNNs speed up CO solvers** As outlined in the previous subsection, current GNN architectures might miss crucial structural patterns in the data. At the same time, more expressive approaches do not scale to large-scale inputs. Moreover, 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. Furthermore, 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 supervised approaches do not give meaningful predictive performance when

23. This specific value has been reported in Table 4 (instance E10M.0) of Helsgaun (2009).

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-offs among these three aspects remains an open challenge for deploying GNNs on CO tasks.

**Programmatic primitives** Existing work in algorithmic reasoning has produced GNN architectures capable of fitting certain classes of iterative algorithms and data structures. That being said, there exist many kinds of reasoning primitives 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 support 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 they 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 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 of the early attempts are discussed in the next section.

## 5. Implementation Frameworks

Nowadays, several well-documented, open-source libraries for implementing custom GNN architectures provide a large set of readily available models from the literature. Notable examples 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. SeaPearl (Chalumeau et al., 2021) is a constraint programming solver guided by reinforcement learning that uses GNNs for representing training instances. Finally, research in algorithmic reasoning has recently

also received a supporting benchmark dataset, CLRS-30 (Veličković et al., 2022a). In this benchmark, graph neural networks execute thirty diverse algorithms outlined in Cormen et al. (2009). Implementations of all relevant data generation pipelines and several popular models in the literature are provided within CLRS, making it a potentially useful starting point for research in the area.

## 6. Conclusions

We gave an overview of the recent applications of GNNs for CO. To that end, we briefly introduced 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 presented recent dual approaches, i.e., those 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 applying GNNs to CO. Finally, we identified 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 help spur novel research results and future avenues.

## Acknowledgements and Disclosure of Funding

Christopher Morris is partially funded by a DFG Emmy Noether grant (468502433) and RWTH Junior Principal Investigator Fellowship under Germany's Excellence Strategy. We thank anonymous referees for their useful comments and helpful suggestions.

## References

R. Abboud, I. Ceylan, and T. Lukasiewicz. Learning to reason: Leveraging neural networks for approximate dnf counting. In *AAAI Conference on Artificial Intelligence*, pages 3097–3104, 2020.

R. Abboud, I. I. Ceylan, M. Grohe, and T. Lukasiewicz. The surprising power of graph neural networks with random node initialization. In *International Joint Conference on Artificial Intelligence*, pages 2112–2118, 2021.

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

S. Ahn, Y. Seo, and J. Shin. Learning what to defer for maximum independent sets. In *International Conference on Machine Learning*, pages 134–144, 2020.

U. Alon and E. Yahav. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint*, abs/2006.05205, 2020.

S. Amizadeh, S. Matusevych, and M. Weimer. Learning to solve circuit-sat: An unsupervised differentiable approach. In *International Conference on Learning Representations*, 2018.

M. C. Angelini and F. Ricci-Tersenghi. Cracking nuts with a sledgehammer: when modern graph neural networks do worse than classical greedy algorithms. *arXiv preprint arXiv:2206.13211*, 2022.

S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Conference on Foundations of Computer Science*, pages 2–11, 1996.

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.

G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

P. Awasthi, A. Das, and S. Gollapudi. Beyond GNNs: A sample efficient architecture for graph problems. In *AAAI Conference on Artificial Intelligence*, 2022.

Y. Bai, H. Ding, K. Gu, Y. Sun, and W. Wang. Learning-based efficient graph similarity computation via multi-scale convolutional set matching. In *AAAI Conference on Artificial Intelligence*, pages 3219–3226, 2020.

A. Bansal, A. Schwarzschild, E. Borgnia, Z. Emam, F. Huang, M. Goldblum, and T. Goldstein. End-to-end algorithm synthesis with recurrent networks: Extrapolation without overthinking. In *Advances in Neural Information Processing Systems*, 2022.

G. Behnke, D. Höller, and S. Biundo. totsat-totally-ordered hierarchical planning through sat. In *AAAI Conference on Artificial Intelligence*, 2018.

N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. 2005.

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.

V. E. Beneš et al. *Mathematical theory of connecting networks and telephone traffic*. Academic press, 1965.

Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning*, 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. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

L. Beurer-Kellner, M. Vechev, L. Vanbever, and P. Veličković. Learning to configure computer networks with neural algorithmic reasoning. In *Advances in Neural Information Processing Systems*, 2022.

B. Bevilacqua, Y. Zhou, and B. Ribeiro. Size-invariant graph representations for graph classification extrapolations. In *International Conference on Machine Learning*, pages 837–851, 2021.

C. Blundell, L. Buesing, A. Davies, P. Veličković, and G. Williamson. Towards combinatorial invariance for Kazhdan-Lusztig polynomials. *Representation Theory*, 2022.

M. Böther, O. Kißig, M. Taraz, S. Cohen, K. Seidel, and T. Friedrich. What's wrong with deep learning in tree search for combinatorial optimization. *arXiv preprint arXiv:2201.10494*, 2022.

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. *arXiv preprint*, 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, 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. A. Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *AAAI Conference on Artificial Intelligence*, pages 3677–3687, 2021.

Q. Cappart, D. Bergman, L.-M. Rousseau, I. Prémont-Schwarz, and A. Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 2022.

F. Chalumeau, I. Coulon, Q. Cappart, and L.-M. Rousseau. SeaPearl: A constraint programming solver guided by reinforcement learning. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 392–409, 2021.

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

T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607, 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.

C. Chi, A. Aboussalah, E. Khalil, J. Wang, and Z. Sherkat-Masoumi. A deep reinforcement learning framework for column generation. *Advances in Neural Information Processing Systems*, 35:9633–9644, 2022.

E. Clarke, M. Talupur, H. Veith, and D. Wang. Sat based predicate abstraction for hardware verification. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 78–92, 2003.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

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. *Advances in Neural Information Processing Systems*, pages 13260–13271, 2020.

R. Csordas and J. Schmidhuber. Improving differentiable neural computers through memory masking, de-allocation, and link distribution sharpness control. *International Conference on Learning Representations*, 2019.

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.

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

A. Davies, P. Veličković, L. Buesing, S. Blackwell, D. Zheng, N. Tomašev, R. Tanburn, P. Battaglia, C. Blundell, A. Juhász, M. Lackenby, G. Williamson, D. Hassabis, and P. Kohli. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887): 70–74, 2021.

L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

A. Deac, P. Bacon, and J. Tang. Graph neural induction of value iteration. *arXiv preprint*, abs/2009.12604, 2020.

A. Deac, P. Veličković, O. Milinkovic, P.-L. Bacon, J. Tang, and M. Nikolic. Neural algorithmic reasoners are implicit planners. *Advances in Neural Information Processing Systems*, 2021.

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.

D. Delahaye, S. Chaimatanan, and M. Mongeau. Simulated annealing: From basics to applications. In *Handbook of metaheuristics*, pages 1–35. Springer, 2019.

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. 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. Ozoliņš, A. Šostaks, M. Apinis, and K. Freivalds. Residual shuffle-exchange networks for fast processing of long sequences. In *AAAI Conference on Artificial Intelligence*, pages 7245–7253, 2021.

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. In *IEEE International Conference on Machine Learning and Applications*, pages 19–24, 2020.

H. Duan, P. Vaezipoor, M. B. Paulus, Y. Ruan, and C. Maddison. Augment with care: Contrastive learning for combinatorial problems. In *International Conference on Machine Learning*, pages 5627–5642, 2022.

A. J. Dudzik and P. Veličković. Graph neural networks are dynamic programmers. In *Advances in Neural Information Processing Systems*, 2022.

R. Durbin and D. Willshaw. An analogue approach to the travelling salesman problem using an elastic net method. *Nature*, 326(6114):689–691, 1987.

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.

V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *arXiv preprint*, abs/2003.00982, 2020.

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

M. Etheve, Z. Alès, C. Bissuel, O. Juan, and S. Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In *CPAIOR*, 2020.

G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson. TreeQN and ATreeC: Differentiable tree-structured models for deep reinforcement learning. In *International Conference on Learning Representations*, 2018.

P. Festa. A brief introduction to exact, approximation, and heuristic algorithms for solving hard combinatorial optimization problems. In *International Conference on Transparent Optical Networks*, pages 1–20, 2014.

M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. *arXiv preprint*, 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. *arXiv preprint*, 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. Ganesh and M. Y. Vardi. On the unreasonable effectiveness of sat solvers, 2020.

V. Garg, S. Jegelka, and T. Jaakkola. Generalization and representational limits of graph neural networks. In *International Conference on Machine Learning*, pages 3419–3430, 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.

M. Gasse, S. Bowly, Q. Cappart, J. Charfreitag, L. Charlin, D. Chételat, A. Chmiela, J. Dumouchelle, A. Gleixner, A. M. Kazachkov, E. Khalil, P. Lichocki, A. Lodi, M. Lubin, C. J. Maddison, C. Morris, D. J. Papageorgiou, A. Parjadis, S. Pokutta, A. Prouvost, L. Scavuzzo, G. Zarpellon, L. Yang, S. Lai, A. Wang, X. Luo, X. Zhou, H. Huang, S. Shao, Y. Zhu, D. Zhang, T. Quan, Z. Cao, Y. Xu, Z. Huang, S. Zhou, B. Chen, M. He, H. Hao, Z. Zhang, Z. An, and M. Kun. The machine learning for combinatorial optimization competition (ml4co): Results and insights. In *NeurIPS 2021 Competitions and Demonstrations Track*, pages 220–231, 2022.

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

D. Georgiev, P. Barbiero, D. Kazhdan, P. Veličković, and P. Liò. Algorithmic concept-based explainable reasoning. In *AAAI Conference on Artificial Intelligence*, pages 6685–6693, 2022.

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.

S. Gold, A. Rangarajan, et al. Softmax to softassign: Neural network algorithms for combinatorial optimization. *Journal of Artificial Neural Networks*, 2(4):381–399, 1996.

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. *arXiv preprint*, 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.

A. Gupta, M. K. Ganai, and C. Wang. SAT-based verification methods and applications in hardware verification. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 108–143, 2006.

P. Gupta, M. Gasse, E. Khalil, P. Mudigonda, A. Lodi, and Y. Bengio. Hybrid models for learning to branch. *Advances in Neural Information Processing Systems*, 33:18087–18097, 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. Hamilton, P. Bajaj, M. Zitnik, D. Jurafsky, and J. Leskovec. Embedding logical queries on knowledge graphs. *Advances in neural information processing systems*, 31, 2018.

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.

P. Hansen, N. Mladenović, J. Brimberg, and J. A. M. Pérez. Variable neighborhood search. In *Handbook of metaheuristics*, pages 57–97. Springer, 2019.

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

Y. He, P. Veličković, P. Lio, and A. Deac. Continuous neural algorithmic planners. In *The First Learning on Graphs Conference*, 2022. URL `https://openreview.net/forum?id=60avttWOMv`.

K. Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European journal of operational research*, 126(1):106–130, 2000.

K. Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1:119–163, 2009.

J. J. Hopfield and D. W. Tank. "Neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

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. *arXiv preprint*, abs/2008.06319, 2020.

A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys*, 50(2):1–35, 2017.

B. Ibarz, V. Kurin, G. Papamakarios, K. Nikiforou, M. Bennani, R. Csordás, A. J. Dudzik, M. Bošnjak, A. Vitvitskyi, Y. Rubanova, A. Deac, B. Bevilacqua, Y. Ganin, C. Blundell, and P. Veličković. A generalist neural algorithmic learner. In *The First Learning on Graphs Conference*, 2022.

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

C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, pages 1–29, 2022.

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. *arXiv preprint*, 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. B. Khalil, C. Morris, and A. Lodi. MIP-GNN: A data-driven framework for guiding combinatorial solvers. In *AAAI Conference on Artificial Intelligence*, 2022.

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. In *International Joint Conference on Artificial Intelligence*, pages 4475–4482, 8 2021.

O. Kramer. Genetic algorithms. In *Genetic algorithm essentials*, pages 11–19. Springer, 2017.

K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *arXiv preprint*, 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.

A. G. Labassi, D. Chételat, and A. Lodi. Learning to compare nodes in branch and bound with graph neural networks. In *Advances in Neural Information Processing Systems*, volume 35, pages 32000–32010, 2022.

M. Laguna. Tabu search. In *Handbook of heuristics*, pages 741–758. Springer, 2018.

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.

A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

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.

H. Lemos, M. Prates, P. Avelar, and L. Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In *IEEE International Conference on Tools with Artificial Intelligence*, pages 879–885, 2019.

Q. Li, Z. Han, and X.-M. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI Conference on Artificial Intelligence*, pages 3538–3545, 2018a.

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. *arXiv preprint*, 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, 2018b.

Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in Neural Information Processing Systems*, 31, 2018c.

D. Liu, A. Lodi, and M. Tanneau. Learning chordal extensions. *Journal of Global Optimization*, 81(1):3–22, 2021.

D. Liu, M. Fischetti, and A. Lodi. Learning to search in local branching. In *AAAI Conference on Artificial Intelligence*, pages 3796–3803, 2022.

A. Lodi. Mixed integer programming computation. In *50 years of integer programming 1958-2008*, pages 619–645. Springer, Berlin, Heidelberg, 2010.

A. Lodi. The heuristic (dark) side of MIP solvers. In *Hybrid metaheuristics*, pages 273–284. Springer, 2013.

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. *arXiv preprint*, abs/1911.04936, 2019.

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

F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208, 2006.

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.

J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 133–182. 2021.

N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

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. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

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.

M. Morabit, G. Desaulniers, and A. Lodi. Machine-learning–based column selection for column generation. *Transportation Science*, 55(4):815–831, 2021.

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.

C. Morris, M. Fey, and N. M. Kriege. The power of the Weisfeiler-Leman algorithm for machine learning with graphs. In *International Joint Conference on Artificial Intelligence*, pages 4543–4550, 2021.

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. *arXiv preprint*, 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.

M. Niepert, P. Minervini, and L. Franceschi. Implicit MLE: backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, pages 14567–14579, 2021.

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 *IEEE Data Science Workshop*, pages 1–5, 2018.

D. Numeroso, D. Bacciu, and P. Veličković. Dual algorithmic reasoning. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=hhvkdRdWt1F`.

E. Ong and P. Veličković. Learnable commutative monoids for graph neural networks. In *The First Learning on Graphs Conference*, 2022.

E. Ozolins, K. Freivalds, A. Draguns, E. Gaile, R. Zakovskis, and S. Kozlovics. Goal-aware neural SAT solver. In *International Joint Conference on Neural Networks*, pages 1–8, 2022.

R. Palm, U. Paquet, and O. Winther. Recurrent relational networks. *Advances in Neural Information Processing Systems*, 31, 2018.

A. Parjadis, Q. Cappart, L.-M. Rousseau, and D. Bergman. Improving branch-and-bound using decision diagrams and reinforcement learning. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 446–455, 2021.

M. B. Paulus, G. Zarpellon, A. Krause, L. Charlin, and C. Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *International Conference on Machine Learning*, pages 17584–17600, 2022.

T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 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.

J.-Y. Potvin and M. Gendreau. *Handbook of Metaheuristics*. Springer, 2018.

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 *AAAI Conference on Artificial Intelligence*, 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. *arXiv preprint*, abs/2011.06069, 2020.

J. Ramanujam and P. Sadayappan. Mapping combinatorial optimization problems onto neural networks. *Information sciences*, 82(3-4):239–255, 1995.

S. Reed and N. De Freitas. Neural programmer-interpreters. *arXiv preprint*, 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.

J.-C. Régin. Global constraints and filtering algorithms. In *Constraint and Integer Programming*, pages 89–135. Springer, 2004.

H. Ren, W. Hu, and J. Leskovec. Query2box: Reasoning over knowledge graphs in vector space using box embeddings. In *International Conference on Learning Representations*, 2019.

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

S. 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. Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468, 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. In *SIAM International Conference on Data Mining*, pages 333–341. SIAM, 2021.

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.

L. Scavuzzo, F. Chen, D. Chételat, M. Gasse, A. Lodi, N. Yorke-Smith, and K. Aardal. Learning to branch with tree mdps. In *Advances in Neural Information Processing Systems*, volume 35, pages 18514–18526, 2022.

M. J. A. Schuetz, J. K. Brubaker, and H. G. Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022.

A. Schwarzschild, E. Borgnia, A. Gupta, F. Huang, U. Vishkin, M. Goldblum, and T. Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. *Advances in Neural Information Processing Systems*, 34:6695–6706, 2021.

B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *AAAI Conference on Artifical Intelligence*, volume 94, pages 337–343, 1994.

D. Selsam and N. Bjørner. Guiding high-performance sat solvers with unsat-core predictions. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing*, pages 336–353, 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.

K. 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.

W. Song, Z. Cao, J. Zhang, C. Xu, and A. Lim. Learning variable ordering heuristics for solving constraint satisfaction problems. *Engineering Applications of Artificial Intelligence*, 109:104603, 2022.

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. *arXiv preprint*, abs/2103.01043, 2021.

H. Sun, W. Chen, H. Li, and L. Song. Improving learning to branch via reinforcement learning. In *Workshop on Learning Meets Combinatorial Algorithms, NeurIPS*, 2020a.

L. Sun, D. Gerault, A. Benamira, and T. Peyrin. Neurogift: Using a machine learning based sat solver for cryptanalysis. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 62–84, 2020b.

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

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

E. D. Taillard and K. Helsgaun. Popmusic for the travelling salesman problem. *European Journal of Operational Research*, 272(2):420–429, 2019.

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.

C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *International Conference on Software Engineering*, pages 178–188, 2007.

P. Vaezipoor, G. Lederman, Y. Wu, C. Maddison, R. B. Grosse, S. A. Seshia, and F. Bacchus. Learning branching heuristics for propositional model counting. In *AAAI Conference on Artificial Intelligence*, pages 12427–12435, 2021.

R. van Driel, E. Demirović, and N. Yorke-Smith. Learning variable activity initialisation for lazy clause generation solvers. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 62–71. Springer, 2021.

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 and C. Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021.

P. Veličković, L. Buesing, M. C. Overlan, R. Pascanu, O. Vinyals, and C. Blundell. Pointer graph networks. *Advances in Neural Information Processing Systems*, 33:2232–2244, 2020.

P. Veličković, A. P. Badia, D. Budden, R. Pascanu, A. Banino, M. Dashevskiy, R. Hadsell, and C. Blundell. The CLRS algorithmic reasoning benchmark. In *International Conference on Machine Learning*, 2022a.

P. Veličković, M. Bošnjak, T. Kipf, A. Lerchner, R. Hadsell, R. Pascanu, and C. Blundell. Reasoning-modulated representations. In *The First Learning on Graphs Conference*, 2022b.

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

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.

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. *arXiv preprint*, 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.

M. Winkenbach, S. Parks, and J. Noszek. Technical proceedings of the amazon last mile routing research challenge. 2021.

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

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

L.-P. Xhonneux, A.-I. Deac, P. Veličković, and J. Tang. How to transfer algorithmic reasoning knowledge to learn new algorithms? *Advances in Neural Information Processing Systems*, 34:19500–19512, 2021.

H. Xu, K.-H. Hui, C.-W. Fu, and H. Zhang. TilinGNN: learning to tile with self-supervised graph neural network. *ACM Transactions on Graphics*, 39(4):129–1, 2020a.

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

K. Xu, J. Li, M. Zhang, S. S. Du, K.-I. Kawarabayashi, and S. Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2020b.

K. Xu, M. Zhang, J. Li, S. S. Du, K.-I. Kawarabayashi, and S. Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2021.

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. *arXiv preprint*, 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. In *International Conference on Machine Learning*, pages 11773–11783, 2021.

W. Yao, A. S. Bandeira, and S. Villar. Experimental performance of graph neural networks on random instances of max-cut. In *Wavelets and Sparsity XVIII*, volume 11138, pages 242–251. SPIE, 2019.

G. Yehuda, M. Gabel, and A. Schuster. It's not what machines can learn, it's what we cannot teach. In *International Conference on Machine Learning*, pages 10831–10841, 2020.

G. Yehudai, E. Fetaya, E. Meirom, G. Chechik, and H. Maron. From local structures to size generalization in graph neural networks. In *International Conference on Machine Learning*, pages 11975–11986, 2021.

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

J. You, Z. Ying, and J. Leskovec. Design space for graph neural networks. In *Advances in Neural Information Processing Systems*, 2020.

W. Zaremba and I. Sutskever. Learning to execute. *arXiv preprint*, 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, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

Z. Zhu, Z. Zhang, L.-P. Xhonneux, and J. Tang. Neural Bellman-Ford networks: A general graph neural network framework for link prediction. *Advances in Neural Information Processing Systems*, 34:29476–29490, 2021.