Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

# Learning-Augmented Combinatorial Solvers: A Solver-Architecture Taxonomy

**Morteza Kimiaei**

kimiaeim83@univie.ac.at

University of Vienna

**Vyacheslav Kungurtsev**

Czech Technical University in Prague

**Research Article**

**Keywords:** Learning-Augmented Solvers, Combinatorial Optimization, Boolean Satisfiability, Conflict-Driven Clause Learning, Lazy Clause Generation, Constraint- Based Solving, Disjunctive Optimization, Machine Learning, Reinforcement Learning, Graph Neural Networks, Learned Heuristics, Neuro-Symbolic Optimization, Hybrid Solvers

**Posted Date:** November 2nd, 2026

**DOI:** https://doi.org/10.21203/rs.3.rs-11270826/v1

**License:** This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

**Additional Declarations:** No competing interests reported.

# Learning-Augmented Combinatorial Solvers: A Solver-Architecture Taxonomy

**Morteza Kimiaei***
*Fakultät für Mathematik, Universität Wien*
*Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria*
*Email:* kimiaeim83@univie.ac.at
*WWW:* http://www.mat.univie.ac.at/~kimiaei

(*Corresponding Author)

**Vyacheslav Kungurtsev**
*Department of Computer Science, Czech Technical University*
*Karlovo Namesti 13, 121 35 Prague 2, Czech Republic*
*Email:* vyacheslav.kungurtsev@fel.cvut.cz

**Abstract.** This paper presents a solver-architecture taxonomy of machine-learning and reinforcement-learning interventions in combinatorial solving. Rather than treating learning-augmented solving as a collection of isolated applications, we organize existing methods according to the solver-level decisions they influence, including propagation, branching, conflict analysis, restart control, structural analysis, and formulation-level reasoning. The discussion focuses on SAT/CDCL, Lazy Clause Generation, and related constraint-based, disjunctive, and graph-structured optimization frameworks, where learned models are increasingly used to guide decisions traditionally governed by handcrafted heuristics. We connect these intervention points to representative learning paradigms, including supervised prediction, reinforcement learning, graph neural networks, transformer-based encoders, and neuro-symbolic reasoning. The resulting taxonomy clarifies where learning can be integrated without compromising solver correctness, where empirical evidence remains limited, and where issues of generalization, interpretability, data generation, and runtime overhead remain open. The paper aims to provide a structured conceptual map for researchers developing hybrid symbolic–statistical solvers for hard combinatorial problems.

**Keywords.**

Learning-Augmented Solvers; Combinatorial Optimization; Boolean Satisfiability; Conflict-Driven Clause Learning; Lazy Clause Generation; Constraint-Based Solving; Disjunctive Optimization; Machine Learning; Reinforcement Learning; Graph Neural Networks; Learned Heuristics; Neuro-Symbolic Optimization; Hybrid Solvers.

# 1 Introduction

Optimization and constraint reasoning play key roles in both Operations Research (OR) and Artificial Intelligence (AI). Many hard combinatorial problems are solved by architectures that combine propagation, search, relaxation, conflict analysis, restart policies, and heuristic decision rules. Boolean Satisfiability (`SAT`), Conflict-Driven Clause Learning (`CDCL`), Lazy Clause Generation (`LCG`), and constraint-based optimization provide important examples of this solver-centric view [3, 8, 16, 32, 37]. These solvers preserve formal correctness through symbolic inference, but their practical performance often depends on heuristic choices such as variable selection, value ordering, clause retention, restart timing, decomposition, and formulation design. Machine Learning (ML) and Reinforcement Learning (RL) are increasingly used to inform these choices by identifying structural patterns in instances, predicting solver behavior, or adapting policies during search. This paper studies such *learning-augmented solvers* from the perspective of solver-level intervention points rather than from the perspective of a single application domain or a single learning model.

## 1.1 Problem Setting

This survey examines how learning-based modules are inserted into classical combinatorial solver pipelines. The focus is not on proposing new benchmarks or new algorithms, but on organizing the literature according to the solver-level decisions that learning modifies. A constraint satisfaction problem may be written as $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, with variables $\mathcal{X}$, domains $\mathcal{D}$, and constraints $\mathcal{C}$; `SAT`, `LCG`, and several disjunctive or graph-structured optimization models can be viewed as related settings in which feasibility, propagation, search, or relaxation decisions must be coordinated. Traditional solvers explore the feasible assignment or relaxation space through deterministic inference and handcrafted heuristics. Learning-augmented solvers instead estimate probabilistic guidance functions $\pi_\theta$, predictive mappings $f_\theta$, or learned scores that influence propagation, branching, restart, clause management, solver selection, or formulation-level reasoning. Therefore, this survey focuses on how learning modifies the main decision points of solver architectures.

## 1.2 Challenges in Learning-Augmented Combinatorial Solving

Several long-standing issues motivate the solver-architecture perspective adopted in this survey. (i) **Scalability.** Classical `CSP`/`SAT` solvers must search exponential spaces, and learned components often have difficulty handling in-

stance structures that differ from those seen during training. (ii) **Interpretability.** Neural approaches do not offer the explicit reasoning steps of rule-based solvers, which complicates verification in sensitive applications. (iii) **Data and Transferability.** Generating labeled solver traces or optimal solutions is costly, limiting the scope of supervised methods and their ability to transfer across domains. (iv) **Integration Consistency.** Hybrid solver designs are frequently constructed on an ad hoc basis, and many lack a principled account of how learning modules interact with symbolic procedures while preserving completeness or soundness. These challenges motivate the topics covered in the survey.

## 1.3 Research Questions

The survey is framed by three questions that delineate the interactions between learning approaches and established methods in constraint optimization. They correspond respectively to (i) the integration of ML and RL techniques into solver pipelines, (ii) the representational and algorithmic frameworks that enable such integration, and (iii) the theoretical and empirical challenges that shape current research directions.

**Q1. Which solver-level decisions have been targeted by ML and RL methods?** This question examines learning-based enhancements to propagation, branching, value selection, conflict analysis, restart policies, clause or nogood management, solver selection, and formulation-level reasoning.

**Q2. Which representations and learning paradigms support these interventions?** It investigates the role of handcrafted features, graph representations, variable–clause embeddings, GNNs, transformer encoders, differentiable logic, reinforcement-guided policies, and hybrid symbolic–statistical architectures.

**Q3. What limits the reliability and transferability of learning-augmented solvers?**

This includes scalability, data generation, generalization across instance distributions, runtime overhead, interpretability, benchmark bias, and the preservation of soundness and completeness when learned components are embedded into symbolic solvers.

These questions determine the structure of the taxonomy and analysis, and help relate established solver theory to current learning-based techniques.

## 1.4 Survey Scope and Organization

This paper is organized as a *solver-architecture taxonomy* and conceptual synthesis of learning-augmented combinatorial solving. It does not aim to be an exhaustive survey of all machine learning methods for constraint programming, SAT, mathematical programming, or combinatorial optimization. Instead, it focuses on where learning is inserted into solver workflows and on how learned components interact with symbolic inference, search, relaxation, and control. The discussion therefore draws selectively from SAT/CDCL, LCG, constraint-based solving, graph-structured optimization, and disjunctive formulations when they illustrate distinct solver-level intervention points. This positioning complements broader surveys on reinforcement learning for combinatorial optimization, graph neural networks for combinatorial reasoning, and constraint programming–machine learning (CP–ML) integration by emphasizing the architectural location and operational role of the learned component.

Section 2 introduces the solver concepts needed for the taxonomy, including constraint representation, SAT, propagation, search, CDCL, and LCG. Section 3 is reframed as background on graph-structured optimization models that motivate message passing, graph embeddings, and GNN-based solver guidance. Section 4 analyzes learning interventions in classical search, including propagation support, branching, conflict analysis, restart control, and structural SAT analysis. Section 5 discusses reinforcement-guided heuristic adaptation. Section 6 treats disjunctive and scheduling-oriented formulations as examples where learned policies interact with formulation structure and resource-exclusion constraints. The later sections synthesize limitations, evaluation issues, and research directions, with emphasis on generalization, soundness, interpretability, and runtime overhead.

## 1.5 Relation to Existing Surveys

Several recent surveys have investigated the interaction between machine learning, constraint reasoning, and combinatorial optimization. These works provide important background for the present paper, but they differ in their primary organizing principles and in the solver paradigms that they emphasize.

Mazyavkina et al. [24] survey reinforcement learning methods for combinatorial optimization. Their focus is on modeling combinatorial optimization problems as sequential decision-making tasks, usually through a Markov decision process with states, actions, rewards, transitions, and a policy to be learned. They classify reinforcement learning approaches according to standard RL design choices, such as value-based and policy-based methods, and discuss their use on canonical combinatorial optimization problems. Thus, their survey is mainly organized around reinforcement learning as a methodological paradigm for combinatorial optimization.

Popescu et al. [29] provide an overview of machine learning techniques in constraint solving. Their survey covers constraint satisfaction, SAT solving, answer set programming, and applications such as configuration, diagnosis, and constraint-based recommendation. They identify several major uses of machine learning in constraint solving, including direct solution search, constraint learning, learning search heuristics or parameters, satisfiability prediction, and algorithm selection. Their contribution is therefore broad with respect to constraint-solving tasks and applications.

Cappart et al. [10] study combinatorial optimization and reasoning with graph neural networks. Their survey is centered on graph representations, message passing, and the role of graph neural networks as either direct solvers or components that enhance exact solvers. They emphasize the inductive bias of graph neural networks for sparse, relational, and permutation-invariant combinatorial structures, and they also discuss the limitations of current graph-based learning approaches.

More recently, Cappart et al. [11] survey the integration of constraint programming and machine learning. Their work is explicitly focused on methods that contain both a CP component and an ML component. In one direction, they review how machine learning can improve the CP toolbox, for example by assisting modeling or accelerating solving. In the other direction, they survey how CP can support machine learning by providing structure, guarantees, explainability, and reasoning capabilities. Their scope is thus centered on the CP–ML interface.

The present paper complements these surveys by adopting a solver-architecture viewpoint. Rather than organizing the literature primarily by the learning method, the application domain, or a single solving paradigm, we organize learning-augmented methods according to the solver-level decision point at which learning is introduced. The main decision points considered here are propagation, branching, conflict analysis, restart control, structural analysis, solver selection, and formulation-level reasoning. This viewpoint is useful for comparing learning interventions across SAT/CDCL, Lazy Clause Generation, constraint-based search, graph-structured optimization, and disjunctive optimization frameworks.

This distinction also clarifies the intended scope of the paper. We do not aim to replace comprehensive surveys on reinforcement learning for combinatorial optimization, machine learning in constraint solving, graph neural networks for combinatorial reasoning, or CP–ML integration. Instead, our goal is to provide a solver-centric map of learning interventions: which component of the solver is modified, what information is learned, how the learned model interacts with the symbolic or optimization engine, and what risks arise in terms of soundness, generalization, interpretability, and computational overhead. This architecture-level perspective is the basis for the taxonomy used in the remainder of the paper.

## 1.6 Our Contributions

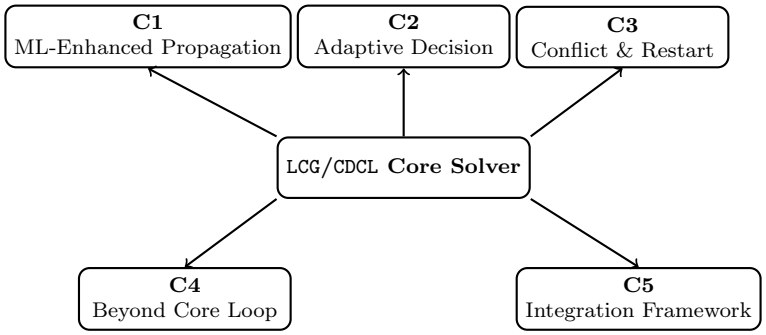



Figure 1: Compact overview of the solver-architecture taxonomy, highlighting five classes of learning intervention in combinatorial solver workflows.

As a survey and conceptual synthesis, this work proposes a taxonomy for understanding how ML and RL methods interact with the internal mechanisms of combinatorial solvers. The taxonomy complements prior surveys by emphasizing the solver component at which learning is introduced, rather than organizing the literature primarily by learning model, application domain, or optimization paradigm. We group the literature around recurring solver-level decisions, including propagation, branching, conflict analysis, restart control, structural analysis, solver selection, and formulation-level reasoning, and describe how learning-based methods have been used to support or modify these decisions in `SAT/CDCL`, `LCG`, constraint-based, graph-structured, and disjunctive optimization settings.

Under this viewpoint, five key ways in which learning influences solver design emerge. These groups, displayed in Figure 1, reflect how earlier work is organized rather than introduce new material. The survey's contributions are presented next:

**C1. ML-Enhanced Propagation.** This survey describes methods that bring statistical models into the propagation phase, including logistic regression for backbone prediction, Monte Carlo polarity estimation, and RL for pruning. These approaches extend classical consistency techniques with information learned from data and draw on earlier work in backbone analysis, abstraction–refinement, and domain-focused pruning.

**C2. Adaptive Heuristic Decision.** We discuss learning-based decision strategies, such as instance classification for solver selection, automated parameter tuning, learning-rate branching methods, and neural or transformer-guided branching. These techniques view branching heuristics as decision rules that adapt to the structure of a given instance.

**C3. Learning-Guided Conflict Analysis and Restart Policies.** Research in this area includes models for predicting clause utility, guiding clause retention, and adjusting restart schedules using supervised learning or RL. Examples range from regression-driven clause scoring to RL-controlled restart policies and neural ranking approaches used in settings such as cryptanalysis. These methods complement the conflict analysis components of `CDCL` and `LCG`.

**C4. Learning Beyond the Core Solver Loop.** ML has also been applied to structural analysis and model-level reasoning, including community detection, phase-transition analysis, symmetry detection, solver selection, and formulation support. We also discuss exact and constraint-based approaches to interpretable model learning, especially optimal decision trees, where solver technology is used to enforce structural constraints, optimize prediction accuracy, and improve interpretability. Representative examples include constraint-programming models with AND/OR search and caching [40], caching branch-and-bound methods based on itemset-mining ideas [1], and dynamic-programming and search algorithms for optimal classification trees [15]. These examples show that learning-augmented solving is not restricted to the inner search loop: it also affects how instances are represented, decomposed, encoded, selected, and interpreted.

**C5. Solver-Architecture Taxonomy.** Finally, we outline a solver-centric taxonomy that compares learning interventions by their operational role, input representation, learning signal, expected benefit, and integration risk. This taxonomy does not merely list applications; it identifies where learning changes solver behavior, where symbolic safeguards remain necessary, and where current empirical evidence is still weak. It thereby maps the design space for future hybrid solvers while making explicit the trade-offs between adaptivity, overhead, generalization, and correctness.

Together, these contributions aim to consolidate a rapidly developing research area and to highlight patterns, connections, and open challenges that guide the design of learning-augmented constraint solvers.

## 1.7 Supplemental Material and Algorithmic Overview

This work is accompanied by a supplemental material, `suppMat.pdf`, which offers full pseudocode, flowcharts, and additional algorithmic material for the ML and RL modules associated with the learning-augmented solver workflows discussed in this paper. The supplement, available at [19, `suppMat.pdf`], includes empirical summaries from the literature and further explanation of the meth-

ods. A full version combining the main paper and the supplement is available as an e-print at [18].

A complete overview of all algorithms used or referenced in this work is provided in Table 1, which summarizes Algorithms 1–18 from the supplemental material.

| **Algorithm** | **Short Title** |
|---|---|
| Algorithm 1 | Advantage Actor–Critic (A2C) |
| Algorithm 2 | Proximal Policy Optimization (PPO) |
| Algorithm 3 | AC-3 Arc Consistency |
| Algorithm 4 | Depth-First Backtracking Search (BTS) |
| Algorithm 5 | Branch-and-Bound (BB) |
| Algorithm 6 | Breadth-First Search (BFS) |
| Algorithm 7 | Depth-First Search (DFS) |
| Algorithm 8 | A* Search Algorithm |
| Algorithm 9 | Generic Lazy Clause Generation (LCG) |
| Algorithm 10 | Generic Conflict-Driven Clause Learning (CDCL) |
| Algorithm 11 | ML-Enhanced Propagation for improving LCG and CDCL |
| Algorithm 12 | ML-Enhanced Heuristic Decision for improving LCG and CDCL |
| Algorithm 13 | ML-Enhanced Conflict Analysis & Backjumping for improving LCG and CDCL |
| Algorithm 14 | ML-Based SAT Structural Analysis and Exact/Constraint-Based Model Learning |
| Algorithm 15 | Variable State Independent Decaying Sum (VSIDS) Branching Heuristic |
| Algorithm 16 | Conflict History-Based (CHB) Heuristic |
| Algorithm 17 | Differentiable Neural Logic Learning |
| Algorithm 18 | Neural-Assisted Relational CSP Solving |

Table 1: All algorithms defined in the Supplemental Material (Algorithms 1–18).

The supplemental material contains extended pseudocode, flowcharts, and implementation-level details for the algorithms summarized in the main paper. The main text is self-contained with respect to the concepts needed for the taxonomy: it defines the solver phases, the learning targets, the relevant graph representations, and the role of the learned guidance functions. The supplement is used only for expanded algorithmic descriptions, additional examples, and detailed derivations that would interrupt the survey narrative.

In the main text, we use the following common solver-phase abstraction. Phase S0 initializes the instance representation, current assignment or relaxation state, solver parameters, and optional learned model. Phase S1 performs propagation or relaxation-based inference and may use learned scores to prioritize constraints, clauses, variables, or graph components. Phase S2 chooses a branch-

ing, value, restart, solver-selection, or formulation-level action using either a handcrafted heuristic or a learned policy $\pi_\theta$. Phase S3 analyzes conflicts, updates clauses or nogoods, backjumps or restarts, and optionally updates learned scores. This S0–S3 abstraction is used only as a unifying description; correctness remains supplied by the underlying symbolic solver, while learned modules act as guidance mechanisms unless explicitly stated otherwise.

# 2 Solver Preliminaries: Constraints, SAT, and Search

This section introduces only the solver concepts needed for the taxonomy developed later. Constraint satisfaction, Boolean satisfiability, propagation, search, and conflict learning provide a common vocabulary for describing where learned components can be inserted into solver workflows. The goal is not to give a comprehensive introduction to constraint programming or SAT, but to identify the architectural mechanisms—such as propagation, branching, conflict analysis, restart, and clause or nogood management—that recur across learning-augmented solvers.

Classical techniques remain essential because they provide correctness guarantees and scalable symbolic inference even when learned models are used for guidance. Throughout the paper, learned components are therefore treated primarily as heuristic or predictive modules that influence solver decisions while the underlying solver maintains feasibility, soundness, or completeness whenever these properties are required.

## 2.1 General Definition of CSPs

CSPs are mathematical formulations consisting of a finite set of variables, each with a discrete domain, and a set of constraints that specify allowable combinations of values. The objective is to find an assignment of values that satisfies all constraints [30, 39].

Formally, a CSP can be defined as a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is a finite set of decision variables, $\mathcal{D} = \{\mathcal{D}(x_1), \mathcal{D}(x_2), \ldots, \mathcal{D}(x_n)\}$ is the collection of domains, where $\mathcal{D}(x_i)$ specifies the possible values of $x_i$ (it is assumed that all domains $\mathcal{D}(x_i)$ are nonempty and that each constraint $c_j$ is well defined over its scope), $\mathcal{C} = \{c_1, c_2, \ldots, c_q\}$ is the set of constraints [30, 39]. Each constraint $c_j$ is defined over a subset $\texttt{scope}(c_j)$ of variables $\mathcal{X}$ and restricts the allowable combinations of values for these variables:

$$c_j \subseteq \prod_{x_i \in \texttt{scope}(c_j)} \mathcal{D}(x_i),$$

where $\prod$ denotes the Cartesian product. Here, $\texttt{scope}(c_j)$ denotes the subset of variables in $\mathcal{X}$ on which constraint $c_j$ depends.

**Set-Theoretic and Logical Formulation.** In the set-theoretic language, each variable $x_i$ takes values from a finite set of symbols or states $\{S_{i1}, S_{i2}, \ldots, S_{iN_i}\}$, i.e.,

$$x_i \in \{S_{i1}, S_{i2}, \ldots, S_{iN_i}\}$$

and each constraint $c_j \in \mathcal{C}$ defines a subset $c_j \subseteq \prod_{x_i \in \texttt{scope}(c_j)} \mathcal{D}(x_i)$ of the corresponding Cartesian product that enumerates all admissible combinations of variable assignments. This representation connects the classical numeric and symbolic perspectives: propositional formulas and finite-domain logical constraints can be represented as set constraints of the above form.

In the **propositional** setting, constraints are Boolean clauses composed of literals $(x_i)$ or $(\neg x_i)$ connected by logical operators $\wedge, \vee, \neg$. In a finite-domain relational setting, quantified or relational constraints can also be represented after grounding over the relevant finite domains; for example,

$$\forall x_i, x_j \, [\, \text{conflict}(x_i, x_j) \Rightarrow x_i \neq x_j \,],$$

which states that two conflicting objects cannot be assigned the same value. This observation is used here only to clarify the modeling connection between logical constraints and finite-domain CSPs; the survey itself focuses on solver-level mechanisms such as propagation, branching, conflict learning, restart control, and learned heuristic guidance.

A solution to a CSP is an assignment $\mathcal{A} : \mathcal{X} \to \bigcup_{i=1}^{n} \mathcal{D}(x_i)$ such that $\mathcal{A}(x_i) \in \mathcal{D}(x_i)$ for all $x_i \in \mathcal{X}$, and for every constraint $c_j \in \mathcal{C}$ the condition $\mathcal{A}(\texttt{scope}(c_j)) \in c_j$ holds. If such an assignment exists, the CSP is called **satisfiable**; otherwise, it is **unsatisfiable**.

Several types of CSPs are commonly distinguished: (i) **finite** CSPs (all domains $\mathcal{D}(x_i)$ are finite), (ii) **infinite** CSPs (at least one $\mathcal{D}(x_i)$ is infinite), (iii) **Boolean** CSPs (all domains $\mathcal{D}(x_i) = \{0, 1\}$), and (iv) **soft** CSPs (constraints incorporate preferences or weights).

CSPs form the foundation for a wide range of combinatorial optimization problems in both OR and AI. When an objective function $f(x)$ is introduced, CSPs extend naturally to discrete optimization problems such as Integer Programming (IP) or Boolean Satisfiability (SAT). These models unify diverse problems—ranging from resource allocation and task scheduling to routing and planning—under a common framework of logical and algebraic feasibility.

Each constraint $c_i$ can equivalently be viewed as a predicate

$$c_i : \prod_{x_j \in \texttt{scope}(c_i)} \mathcal{D}(x_j) \to \{\text{True}, \text{False}\},$$

which evaluates whether the tuple of variable assignments satisfies the relation.

**Embedding into Optimization Frameworks.** A CSP can also be expressed as a feasibility version of an Integer Linear or Nonlinear Program. That is, the goal is to find any $x$ satisfying all constraints, without explicitly optimizing an objective function. Let

$$x = (x_1, \ldots, x_n), \quad x_i \in \mathcal{D}(x_i) \subseteq \mathbb{Z} \text{ or } \{0, 1\}.$$

Each constraint $c_i(x)$ can be represented by a linear or nonlinear relation, for example

$$c_i(x) \equiv g_i(x) \leq 0 \quad \text{or} \quad h_i(x) = 0,$$

so that the CSP becomes the feasibility problem

$$\text{Find } x \in \mathcal{D}(x_1) \times \cdots \times \mathcal{D}(x_n) \text{ such that } c_i(x) = \text{True}, \quad \forall i \in [q].$$

When an objective function $f(x)$ is added, this formulation reduces to the INLP/ILP forms introduced below. Thus, CSPs provide a modeling layer that bridges symbolic reasoning and numerical optimization.

**Connection to Boolean Satisfiability.** Among the many subclasses of CSPs, the Boolean case plays a central theoretical and practical role. A special case arises when all variables are Boolean and all domains are $\{0, 1\}$, with each constraint corresponding to a logical clause in **Conjunctive Normal Form** (CNF). This yields the canonical **Satisfiability Problem (SAT)**, which asks whether there exists an assignment $x \in \{0, 1\}^n$ such that all clauses in a Boolean formula are satisfied. Even finite CSPs are NP-complete in general, motivating heuristic, learning-based, and hybrid optimization strategies. Modern SAT solvers extend classical CSP reasoning by incorporating efficient propagation, branching, and conflict-learning mechanisms—topics introduced in the next section.

## 2.2 Integer, Boolean, and Logical Formulations

We begin by presenting a comprehensive set of problem formulations that are discrete or mixed continuous—discrete in their decisions. Many distinct formulations are computationally equivalent, but their chosen representations, integers, sets, or logical variables, offer structural insights that facilitate specialized solution methods.

Let

$$\mathbf{X} := \{x \in \mathbb{R}^n \mid \underline{x} \leq x \leq \overline{x}\} \quad \text{with } \underline{x}, \overline{x} \in \mathbb{R}^n \ (\underline{x} < \overline{x}), \tag{1}$$

and define the Integer Nonlinear Programming (INLP) problem:

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & x \in C_{\text{int}}, \end{array} \tag{2}$$

where $f : C_{\text{int}} \subseteq \mathbf{X} \to \mathbb{R}$ and

$$C_{\text{int}} := \{x \in \mathbf{X} \mid g(x) = 0,\ h(x) \leq 0,\ x_i \in s_i\mathbb{Z},\ i \in [n]\}. \tag{3}$$

where $s_i > 0$ is an integer scaling factor that defines the discretization granularity for variable $x_i$, $[n]$ denotes $\{1, 2, \ldots, n\}$, and

$$g(x) = (g_1(x), \ldots, g_p(x)), \quad h(x) = (h_1(x), \ldots, h_q(x))$$

are the real-valued (possibly non-convex) equality and inequality constraint functions $g_k : C_{\text{int}} \subseteq \mathbf{X} \to \mathbb{R}$ for $k \in [p]$ and $h_j : C_{\text{int}} \subseteq \mathbf{X} \to \mathbb{R}$ for all $j \in [q]$.

If all functions $f$, $g_k$, and $h_j$ are linear, we obtain the Integer Linear Programming (ILP) problem:

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & x \in C_{\text{inl}}, \end{array} \tag{4}$$

with

$$C_{\text{inl}} := \{x \in \mathbf{X} \mid Ax = b,\ Bx \leq d,\ x_i \in s_i\mathbb{Z},\ i \in [n]\}. \tag{5}$$

Here, $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^m$, $d \in \mathbb{R}^p$, and $c \in \mathbb{R}^n$. Structural combinatorial optimization focuses on exploiting properties of $C_{\text{int}}$ or $C_{\text{inl}}$, such as polyhedral geometry, matroidal constraints, or network structure.

When such a structure is absent, computational complexity escalates rapidly, motivating ML and RL methods that detect latent patterns and guide heuristic search.

**Constraint Optimization.** A pure `CSP` specifies a feasibility problem—finding any assignment that satisfies all constraints. When an objective (cost) function $f(x)$, defined over complete assignments of variables, is introduced, the model becomes a **Constraint Optimization Problem**:

$$\min_{x \in \mathcal{D}(x_1) \times \cdots \times \mathcal{D}(x_n)} f(x) \quad \text{s.t.} \quad c_i(x) = \text{True},\ \forall i \in [q].$$

This generalization subsumes models such as ILP, `MaxSAT` (the optimization variant of `SAT`, in which the goal is to satisfy the maximum number or total weight of clauses), and weighted `CSPs`, and bridges symbolic reasoning with numerical optimization—forming the backbone of modern `CSP` frameworks.

## 2.3 Constraint Propagation and Search Paradigms

Classical CSP solving relies heavily on two complementary mechanisms—constraint propagation and systematic search. Techniques such as arc consistency, backtracking, and branch-and-bound exploit constraint structure to prune infeasible regions of the search space. Constraint propagation incrementally reduces variable domains through logical inference, while search procedures explore partial assignments guided by heuristics such as the minimum remaining values and degree-based ordering.

**Constraint Consistency.** A constraint $c_j$ is *arc consistent* if, for every variable $x_i \in \texttt{scope}(c_j)$ and every value $v \in \mathcal{D}(x_i)$, there exists an assignment of values to the remaining variables in $\texttt{scope}(c_j)$ such that $c_j$ is satisfied. Enforcing arc consistency iteratively prunes infeasible values from domains, thus reducing $\prod_i |\mathcal{D}(x_i)|$ without exploring all assignments. This process underlies classical propagation algorithms such as AC-3 [31, Ch. 5.2] and their higher-order extensions. AC-3 refers to a classical algorithm for enforcing Arc Consistency in CSPs by iteratively pruning inconsistent values from variable domains.

While these paradigms are complete and theoretically grounded, they face scalability bottlenecks in dense, high-dimensional constraint spaces, where combinatorial interactions render purely symbolic inference intractable. This motivates the integration of data-driven heuristics—learned from instance distributions—into branching, propagation, and restart policies, as explored in later sections.

Recent advances in ML and RL have shown promising results in this direction. Learned heuristics can adapt variable selection, prioritize constraint activation, and approximate propagation functions, effectively transferring knowledge from solved CSP instances to improve performance on unseen problems. Such hybrid paradigms combine symbolic search precision with data-driven adaptability, forming the foundation for the next generation of scalable CSP solvers.

**Computational Complexity.** Even for binary finite-domain CSPs, the decision version of the satisfiability problem is NP-complete. Consequently, propagation and search procedures such as backtracking and branch-and-bound exhibit exponential worst-case runtime in the number of variables $|\mathcal{X}|$. These complexity limits motivate the development of heuristic and learning–based strategies that exploit structural regularities or instance distributions to achieve practical scalability.

These classical mechanisms form the operational backbone of modern LCG/CDCL solvers, which extend these ideas with clause learning, restart policies, and hybrid heuristic control.

## 2.4 Classical Search Procedures in CSP Solving

Classical CSP solving can be interpreted as a graph–search problem over the space of partial assignments. Each node in the search tree corresponds to a partial assignment $\mathcal{A}_p \subseteq \mathcal{A}$, and each edge represents the assignment of a new variable value consistent with the constraints so far. Let $\mathcal{A}_p(x_i) = v$ denote that variable $x_i$ has been assigned value $v$.

Formally, we define the *search tree* as

$$T = (V_T, E_T), \qquad V_T = \{\mathcal{A}_p \mid \mathcal{A}_p : \mathcal{X}' \subseteq \mathcal{X},\, \mathcal{A}_p(x_i) \in \mathcal{D}(x_i)\},$$

where $(\mathcal{A}_p, \mathcal{A}'_p) \in E_T$ if $\mathcal{A}'_p = \mathcal{A}_p \cup \{(x_i, v)\}$ for some unassigned $x_i$ and $v \in \mathcal{D}(x_i)$ consistent with all $c_j \in \mathcal{C}$. A leaf node represents either a conflict (dead end) or a full assignment $\mathcal{A}$.

Classical search algorithms (Algorithm 3-10, listed in Table 1) form the algorithmic backbone of CSP solving. They provide systematic procedures for exploring the combinatorial search space, applying propagation and pruning rules to reduce infeasible regions. Detailed pseudocode and procedural descriptions of these algorithms are provided in [19, Section 3.1].

## 2.5 DisP and Logical Constraint Systems

DisP provides a unifying logical framework for representing nonconvex combinations of constraints arising in complex CSPs. A disjunctive set is defined as a finite union of polyhedra

$$S = \bigcup_{k=1}^{q} P_k, \quad P_k = \{x \in \mathbb{R}^n : A^{(k)}x \geq b^{(k)}\},$$

where each $P_k$ corresponds to one logical clause $(A^{(k)}x \geq b^{(k)})$. This representation captures constraints of the form

$$(A_1 x \geq b_1) \;\vee\; (A_2 x \geq b_2) \;\vee\; \cdots \;\vee\; (A_q x \geq b_q),$$

which are not directly expressible in a single convex formulation. Following Balas [4], the convex hull $\mathrm{conv}(S)$ admits a compact extended formulation via *lifting and projection*, introducing auxiliary variables $z_k$. A schematic representation of the lifted constraints can be written as

$$A^{(k)}x - b^{(k)}z_k \geq 0, \quad \sum_k z_k = 1, \quad z_k \geq 0,$$

following the convexification principle of DisP. This convexification principle bridges logical disjunctions with linear relaxations, enabling LCG and CDCL

solvers to reason over unions of feasible regions. In CSP terms, such disjunctions generalize Boolean connectives: $\wedge$ (conjunction) corresponds to intersection, $\vee$ (disjunction) to union, and $\neg$ (negation) to complement in the feasible set algebra.

The convex hull $\text{conv}(S)$ serves as the tightest convex relaxation of a disjunctive set. In integer programming, this leads to the generation of *disjunctive cuts* that iteratively approximate the integer hull. Such cuts, derived via lift-and-project or intersection-cut methods, can be embedded in modern CSP solvers to strengthen their linear relaxations without explicit enumeration of disjunctions. This connection clarifies how logical inference and convex analytic methods can operate together to ensure feasibility in models with both discrete and continuous variables.

A simple example arises in the job-shop scheduling constraint

$$(s_i + p_i \leq s_j) \ \vee \ (s_j + p_j \leq s_i),$$

which enforces a non-overlap condition between two operations $i$ and $j$ on the same machine. Its disjunctive representation corresponds to two polyhedra in $(s_i, s_j)$-space, and its convexified relaxation provides linear inequalities linking $s_i$ and $s_j$. These relaxations serve as valid cuts in many scheduling and sequencing formulations and clarify how logical precedence constraints correspond to geometric representations.

In hybrid learning–based solvers, disjunctive constraints are embedded into neural propagation modules that estimate clause activation probabilities. Given a learned clause embedding $\phi_k = \text{Enc}_\theta(A^{(k)}, b^{(k)})$, a soft assignment $\hat{z}_k = \sigma(W_\phi \phi_k)$, where $W_\phi \in \mathbb{R}^{1\times d}$ approximates the binary selector $z_k$, allowing gradient-based relaxation of disjunctive logic. This perspective unifies logical CSPs, integer programming relaxations, and neural constraint embeddings into a single differentiable reasoning framework. Here, $\text{Enc}_\theta$ denotes a neural encoder.

The geometric-logical framing provided by DisP also clarifies its connection to the graph-structured and neural optimization techniques explored later in the paper.

# 3 Graph-Structured Optimization Models as Learning Substrates

Network flow theory forms one of the foundational pillars of OR, concerned with the movement of commodities, information, or resources through interconnected systems. A network is typically modeled as a graph $G = (V, L)$ where vertices $V$ represent entities or locations and edges $L$ represent connections between them, each associated with a capacity, cost, or direction of flow [2].

This section introduces the mathematical structure behind the classical network flow models, which express global routing or allocation decisions through local conservation laws. These formulations capture how limited resources move through a network under physical or logical constraints, providing a geometric and algebraic foundation for many combinatorial optimization problems. They also highlight how fundamental flow constraints mirror the message-passing and propagation mechanisms later used in ML- and RL-enhanced solvers, making network flows both analytical benchmarks and conceptual precursors to modern GNN and RL frameworks operating on graph-structured data. Detailed formulations of canonical flow and disjunctive optimization problems—including traffic routing, wireless spectrum allocation, energy-efficient design, data-center load balancing, network security, transport routing, and machine configuration—are provided in [19, Section 6], where each model is expressed as a linear or mixed-integer program illustrating the conservation, capacity, and disjunction principles that underlie the hybrid optimization frameworks developed in later sections.

Figure 2 summarizes the classical families of Network Flow and Disjunctive Optimization Models, introduced in this section. It makes explicit the relationships between continuous flow-based formulations (such as traffic routing, spectrum allocation, and energy-aware design) and combinatorial disjunctive models (including route selection and machine configuration). These problem classes jointly underpin the hybrid ML and RL frameworks developed in the remainder of the paper. The recursive disjunctive structure underlying these models reveals a duality between spatial convexification and temporal decomposition, providing the foundation for convergence analysis and learned policy design.

`DynP` designates dynamic disjunctive programming, in which time-indexed `DisP` models are extended with adaptive control. The resulting framework captures planning and scheduling problems as instances of dynamic disjunctive optimization. Classical disjunctive graphs capture resource and precedence constraints, while `DynP` and RL introduce temporal adaptation and feedback. Graph-based neural architectures further extend these formulations by embedding combinatorial structure directly into differentiable learning models, enabling scalable and adaptive decision-making.

The combination of disjunctive logic and learning–based control is applicable across a broad spectrum of industrial and scientific domains. In manufacturing, it governs job-shop and flexible-cell scheduling under uncertain machine availability. In logistics, it supports dynamic fleet and routing management, integrating sequential rescheduling decisions with network constraints. In energy and process systems, it enables hybrid operational strategies that switch between discrete operating modes based on stochastic conditions. Risk-sensitive variants, based on distributional RL, ensure robustness against disturbances and improve reliability in safety-critical environments. Table 2 summarizes representative learning–enhanced approaches that combine disjunctive logic, RL,

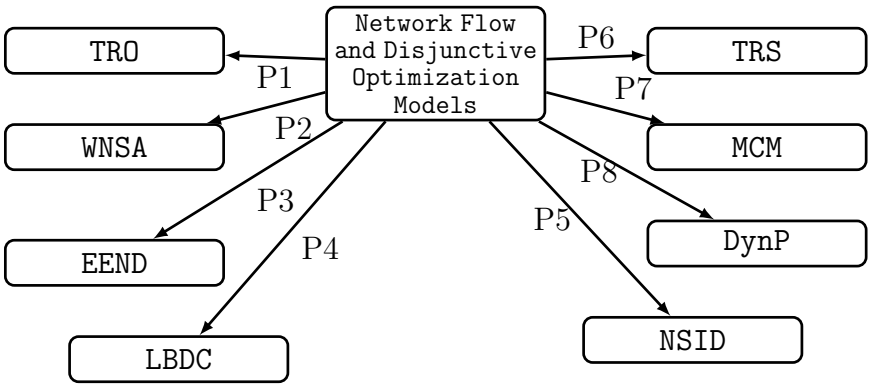



Figure 2: Overview of classical Network Flow and Disjunctive Optimization Models (P1–P8). TRO–Traffic Routing, WNSA–Wireless Spectrum Allocation, EEND–Energy-Efficient Network Design, LBDC–Load Balancing in Data Centers, NSID–Network Security and Intrusion Detection, TRS–Transport Route Selection, MCM–Machine Configuration in Manufacturing, and DynP–Dynamic Disjunctive Programming.

and neural inference, highlighting the diversity of architectures and optimization objectives used in practice.

Table 2: Representative learning-based frameworks for disjunctive scheduling and planning.

| **Approach** | **Main Features and Formulation Highlights** | **Reference** |
|---|---|---|
| GNN–PPO Framework | Encodes disjunctive graph $\mathcal{G} = (V, C \cup D)$; learns dispatching policy $\pi_\phi(a_t \mid s_t)$ for makespan reduction | [28] |
| Attention–Based Deep RL | Transformer encoder captures long-range dependencies; pointer network decoder generates schedules sequentially | [12] |
| Dynamic GNN–RL | Online adaptation to evolving graphs $\mathcal{G}_t$ with stochastic job arrivals and breakdowns; continual policy updates via PPO | [22] |
| Distributional RL (Risk-Aware) | Incorporates stochastic return distributions and CVaR-based objectives for robust scheduling decisions | [26] |
| Differentiable Logic Models | Neural inference of logical constraints and clause activations within disjunctive or ILP formulations | [9] |

In summary, DisP provides the logical backbone for representing alternative actions and resource exclusivity, whereas RL and neural architectures supply adaptive policies that exploit these structures dynamically. Together, they form a general computational paradigm for planning and scheduling under uncertainty—linking symbolic reasoning, optimization, and learning. The next section builds upon these principles by integrating neural architectures directly into SAT and CSP solvers, extending the hybrid logical–learning framework beyond scheduling to general constraint satisfaction.

# 4 Learning for Classical Search

Classical search-based SAT solvers operate on a canonical NP-complete problem that underpins a broad range of combinatorial optimization and logical reasoning methods. They form the computational backbone for reasoning over Boolean formulas and underpin numerous applications in verification, planning, and automated deduction.

Direct solution-learning methods, including the formal CSP solution operator and its learning formulations, are detailed in [19, Section 4]. These foundations provide complementary background for the learning–based enhancements

of classical search discussed in this section.

This section shows how learning modules can be embedded within the principal phases of a conventional SAT solver so that heuristic choices become adaptive rather than fixed.

Each operation—propagation, decision-making, conflict analysis, and restart—can be viewed as a learnable decision process that benefits from data or experience. By learning to guide these phases, solvers can recognize recurring structural patterns and adjust their behavior dynamically. Integrating ML and RL in this way lays the groundwork for self-improving symbolic systems that preserve the logical soundness of classical solvers while gaining the adaptability and generalization power of statistical models. In this view, SAT solving reduces to a succession of reasoning decisions that can be informed by learning, a perspective that underpins the hybrid architectures developed in subsequent sections.

## 4.1 Technical Formulation of the SAT Problem

SAT solvers transform logical reasoning tasks into checking the satisfiability of Boolean formulas expressed in **Conjunctive Normal Form** (CNF):

$$\phi = \bigwedge_{i=1}^{m} \left( \bigvee_{j=1}^{n} l_{ij} \right),$$

where each literal $l_{ij} \in \{x_j, \neg x_j\}$, either a variable $x_j$ or its negation $\neg x_j$, $\bigwedge$ denotes the logical AND operator, and $\bigvee$ denotes the logical OR operator. The goal is to determine whether there exists an assignment $x \in \{0,1\}^n$ such that $\phi(x) = 1$.

From the perspective of constraint reasoning, the SAT problem is a special case of the Constraint Satisfaction Problem (CSP) defined earlier as the triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. In the Boolean setting, all domains are $\mathcal{D}(x_i) = \{0,1\}$, and each constraint $c_j \in \mathcal{C}$ corresponds to a clause in the CNF formula. A solution to SAT is therefore an assignment

$$\mathcal{A} : \mathcal{X} \to \{0,1\}, \quad \text{such that} \quad \mathcal{A}(\texttt{scope}(c_j)) \in c_j, \ \forall c_j \in \mathcal{C}.$$

If no such assignment exists, the problem is **unsatisfiable** (=UNSAT).

From an optimization viewpoint, CSPs and their Boolean subclass SAT can be embedded into ILP/NILP frameworks by encoding logical clauses as linear or nonlinear feasibility constraints, as discussed in the previous section. In this sense, SAT provides a foundational model for logical reasoning and a natural interface through which symbolic methods interact with numerical optimization.

The following sections survey how ML and RL methods have been embedded into classical SAT/CDCL solvers at each algorithmic phase.

## 4.2 Backbone Estimation for SAT Solvers

While the general formulation of the SAT problem was introduced earlier, modern learning–based solvers exploit structural regularities within formulas to guide search. One key concept is that of **backbone variables**, which represent literals that take the same value in every satisfying assignment.

**Backbone Variables.** This paragraph defines backbone variables and logistic Monte Carlo prediction for initial polarity estimation in SAT solving. Given a satisfiable formula $\phi$, the set of backbone variables is

$$\text{Backbone}(\phi) = \big\{\, x_i \;\big|\; \forall\, \mathcal{A}_1, \mathcal{A}_2 \in \texttt{SAT}(\phi),\ \mathcal{A}_1(x_i) = \mathcal{A}_2(x_i) \big\}.$$

Backbone variables correspond to the assignments that remain fixed in all models of $\phi$. Approximating their polarity before search can significantly restrict the branching space and lower the conflict rate during propagation.

**Backbone Prediction via Monte Carlo Sampling.** Following [41], a logistic regression model is trained to estimate the probability that assigning a variable $x_i = 1$ in a partially fixed subformula leads to satisfiability.

For each variable $x_i$, the feature vector $\mathbf{z}_i^{(m)}$ is constructed from local graph statistics of the formula $\phi_m$, e.g.,

$$\begin{aligned}\mathbf{z}_i^{(m)} \quad = \quad & \big[\deg(x_i),\ \#\text{clauses}(x_i),\ \#\text{positive}(x_i),\ \#\text{negative}(x_i),\\ & \text{clause_ratio}(x_i),\ \ldots\big].\end{aligned}$$

These features are normalized and fed into the logistic model

$$p_i^{(m)} = f_\theta(\mathbf{z}_i^{(m)}) = \frac{1}{1 + e^{-(\mathbf{v}^\top \mathbf{z}_i^{(m)} + c)}},$$

which outputs the probability that the simplified subformula $\phi_m$ remains satisfiable when $x_i = 1$.

In each Monte Carlo trial, a random subset $\mathcal{V}_m \subset \mathcal{X}$ is fixed to random truth values, yielding $\phi_m$. The subset $\mathcal{V}_m$ is drawn uniformly at random, and the number of trials is denoted by $M$. Averaging the results yields the empirical satisfiability confidence

$$\bar{p}_i = \frac{1}{M} \sum_{m=1}^{M} p_i^{(m)},$$

and the variable's preferred initial polarity

$$x_i^{(0)} = \begin{cases} 1, & \bar{p}_i > 0.5, \\ 0, & \text{otherwise.} \end{cases}$$

This provides a probabilistic backbone estimate used to initialize the solver's assignments during preprocessing and propagation.

This method casts backbone detection as a probabilistic inference problem. The logistic model offers a differentiable estimate of satisfiability likelihood, while Monte Carlo averaging stabilizes predictions by sampling diverse partial assignments. Subsequent RL-guided propagation and branching (which are discussed in [19, Section 3.2–Algorithms 11-14]) further refine these initial polarity estimates through online interaction with the solver.

## 4.3 Classical Algorithms and Related Solvers

This section reviews the generic flowcharts for **Lazy Clause Generation (`LCG`)** and **Conflict-Driven Clause Learning (`CDCL`)** solver structures, including initialization, propagation, branching, and conflict analysis (see Figures 3–4). Their ML-enhanced variants are summarized in this section and expanded in [19, Algorithms 11-14]. A detailed discussion of representative software implementing `LCG` and `CDCL` algorithms is given in [19, Algorithms 9–10].

State-of-the-art `SAT` solvers follow one of two main algorithmic designs: (i) `LCG`, a hybrid method integrating `CSP`-style propagation with `SAT`-style clause learning. (ii) `CDCL`, a purely Boolean approach operating on `CNF` representations. We next present generic frameworks for both approaches in Figures 3–4.

Figure 3 is a flowchart of the `LCG` procedure. It describes the iterative loop of constraint propagation, heuristic decision-making, and conflict analysis, which continues until it either finds a satisfying assignment or determines that none exists.

Figure 4 shows the corresponding flowchart for the `CDCL` algorithm. Although its high-level structure resembles that of `LCG`, the two differ primarily in the propagation step: `CDCL` performs Boolean unit propagation over a `CNF`-encoded formula, whereas `LCG` integrates `SAT` solving with richer `CSP` domain propagation.

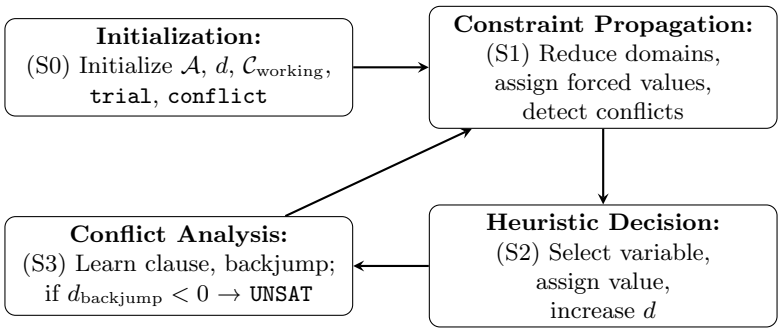



Figure 3: A simplified overview of the generic LCG ( [19, Algorithm 9]) solving cycle using the parameters: $\mathcal{A}$ (current partial assignment), $d$ (decision level), $\mathcal{C}_{\text{working}}$ (working constraint set), trial (assignment record), conflict (conflict flag), and $d_{\text{backjump}}$ (backjump level). (S1)–(S3) repeat until all constraints in $\mathcal{C}_{\text{working}}$ are satisfied or a root-level conflict is detected.

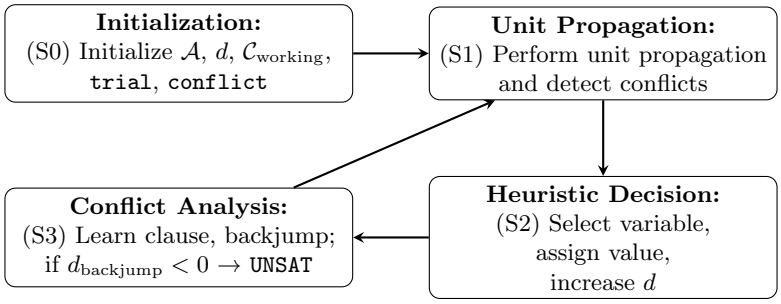



Figure 4: A simplified overview of the generic CDCL ( [19, Algorithm 10]) solving cycle using the parameters: $\mathcal{A}$ (current partial assignment), $d$ (decision level), $\mathcal{C}_{\text{working}}$ (working clause set), trial (assignment record), conflict (conflict flag), and $d_{\text{backjump}}$ (backjump level). (S1)–(S3) repeat until $\mathcal{C}_{\text{working}} = \emptyset$

## 4.4 NNs as Enhancers for CSP Solvers

This section outlines how ML and RL methods have been incorporated into the propagation, decision, and conflict-analysis phases of LCG/CDCL solvers. It is directly underlined by Algorithm 11 (steps: $S0_{11}$–$S3_{11}$), Algorithm 12 (steps: $S0_{12}$–$S3_{12}$), Algorithm 13 (steps: $S0_{13}$–$S3_{13}$), and Algorithm 14 (steps: $S0_{14}$–$S3_{14}$) in [19, Section 3.2]. Figure 5 is a simple flowchart of Algorithm 11, which illustrates how the logistic model predicts backbone literals while an RL policy prunes infeasible domains during constraint propagation. Figure 6 is a simple flowchart of Algorithm 12, which shows adaptive branching, where ML classification and bandit-based or transformer-guided policies dynamically select

variables and polarities. Figure 7 is a simple flowchart of Algorithm 13, which depicts conflict analysis enhanced by ML clause-utility prediction and RL-based restart scheduling to improve solver efficiency. Figure 8 is a simple flowchart of Algorithm 14, which visualizes meta-level learning. In this algorithm, ML models analyze `SAT` structure and construct interpretable decision-tree models via `SAT` encodings.

Using the S0–S3 solver-phase abstraction introduced above, a wide range of ML and RL techniques can be aligned with different stages of the `LCG` and `CDCL` pipelines for `CSP`/`SAT` solving. These include backbone prediction, instance classification, automatic parameter tuning, learning-based branching, polarity prediction, clause-utility estimation, and RL-driven restart strategies. Table 3 summarizes these methods by aligning them with the corresponding solver steps (S1–S3) of the `LCG` and `CDCL` algorithms shown in Figures 3–4, and also lists learning approaches used beyond the core loop. This table focuses on classical ML/RL enhancements within modern solvers, while end-to-end neural architectures such as `NeuroSAT` [34], `NeuralSAT` [33], and `SATFormer` [36] are discussed separately as neural–symbolic extensions that operate directly on clause–literal structures; for more details, see [19, Section 3.2].

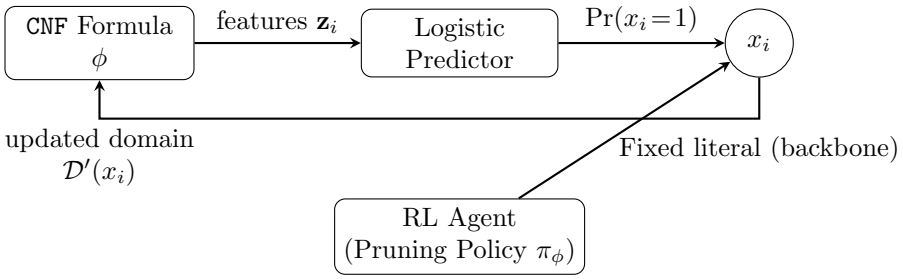



Figure 5: ML-enhanced propagation: the logistic model predicts backbone literals, while an RL agent prunes infeasible branches to accelerate propagation.

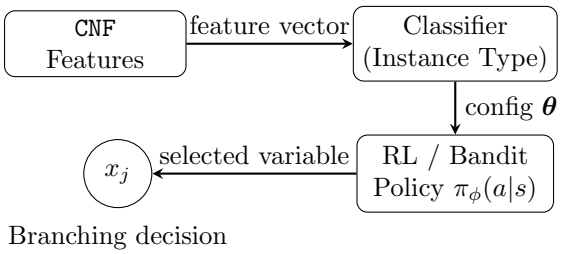



Figure 6: Adaptive branching via ML classification and RL/bandit decision policies. The model selects the next variable and polarity based on learned solver statistics.

Table 3: ML/RL Enhancements in LCG and CDCL Solver Steps

| **Step** | **ML/RL Techniques** | **References** |
|---|---|---|
| (S1) in Figures 3–4: Propagation | • Backbone prediction via logistic regression & Monte Carlo;<br>• RL–guided pruning in domain-specific encodings;<br>• Hybrid abstraction–refinement using ILP relaxations guided by ML | [41]<br>[23]<br>[13] |
| (S2) in Figures 3–4: Heuristic Decision | • Instance classification via CNF features;<br>• Automated parameter tuning with ML;<br>• Learning–based branching (LRB, SGD, etc.);<br>• Classifier-driven branching;<br>• Transformer-based variable selection;<br>• Polarity prediction (default truth values) | [14]<br>[7]<br>[20]<br>[6]<br>[35]<br>[41] |
| (S3) in Figures 3–4: Conflict Analysis & Backjumping | • ML-guided clause utility prediction;<br>• RL-inspired restart policies;<br>• domain-specific RL guidance (timetabling); ML-enhanced SAT solvers in cryptanalysis (NeuroGIFT) | [20]<br>[23]<br>[38] |
| Beyond the Core Loop | ML analysis of community-structured SAT thresholds; exact and constraint-based learning of optimal decision trees (XAI) | [1]<br>[15]<br>[5]<br>[27]<br>[40] |

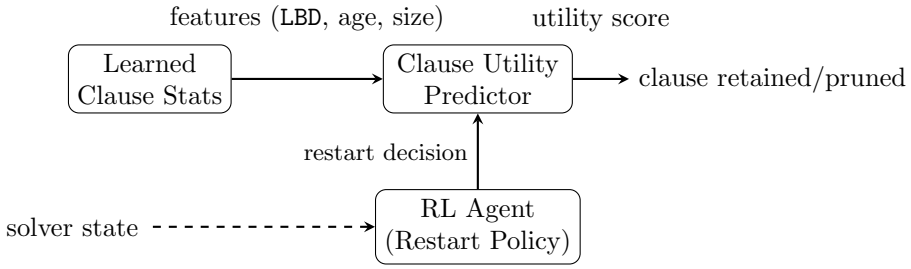



Figure 7: Conflict analysis enhanced by ML-based clause evaluation and RL-guided restart scheduling. LBD (Literal Block Distance) is a metric for evaluating clause quality during conflict learning that is discussed in [19, Section 3.2].

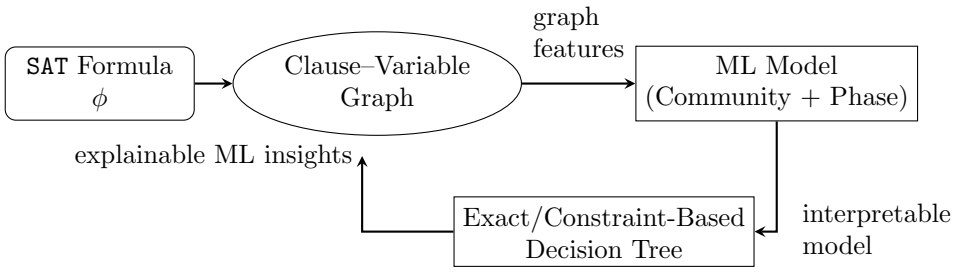



Figure 8: Beyond the solving loop: ML models extract community structure and support interpretable decision-tree learning through exact and constraint-based formulations.

# 5 Learning for Heuristic Search

Heuristic search plays a central role in scaling CSP and SAT solvers by assigning priorities to candidate assignments and steering the search toward regions that are more likely to contain a solution. Branching order, polarity choices, and restart timing all depend on these heuristics, and small changes can significantly affect solver performance. This section discusses how traditional heuristics can be strengthened, or even replaced, by learning-based approaches that adapt as the solver runs. A key point is that heuristic rules can be learned from solver data, allowing them to adjust based on regularities observed during the search. By viewing branching and restart choices as policy-learning tasks, a solver can adjust these decisions based on information gathered during the search. Learning-based heuristics introduce adaptivity into the solver's decision process while retaining the completeness guarantees of the symbolic backbone. They connect classical scoring schemes such as VSIDS and CHB with modern

methods that use supervised or reinforcement learning to guide variable and clause selection, as described in [19, ($S2_{12}$) of Algorithm 12].

## 5.1 Policy-Based Heuristic Generation

Classical solvers employ fixed heuristics such as Variable State Independent Decaying Sum (`VSIDS`) [25] and Conflict History-Based (`CHB`) [21]. Learning-based approaches reinterpret these scoring functions as adaptive policies $\pi_\phi(a_t \mid s_t)$ over solver states $s_t = (\mathcal{A}_t, \mathcal{C}^t_{\text{working}})$. The `VSIDS` heuristic prioritizes variables by assigning and updating activity scores according to their frequency in recently learned clauses, as described in [19, Algorithm 15]. The `CHB` heuristic ranks variables based on their historical success in resolving conflicts, maintaining an effectiveness score that reflects each variable's contribution to past conflict resolutions, as detailed in [19, Algorithm 16].

**Policy-Based Extension.** A neural policy replaces hand-crafted scoring by mapping solver states to action probabilities:

$$\pi_\phi(a_t \,|\, s_t) = \text{softmax}\big(W_\phi \, g_\phi(s_t)\big),$$

where $g_\phi(s_t)$ encodes variable-level features (e.g., degree, activity, recent conflict count). Actions $a_t$ correspond to selecting a variable and its polarity. Training the policy to imitate `VSIDS`/`CHB` decisions via supervised imitation or to maximize downstream reward connects heuristic scoring to adaptive policy generation.

## 5.2 RL for Heuristic Adaptation

Heuristic adaptation can be formalized as a Markov Decision Process (MDP)

$$\mathcal{M}_{\text{heur}} = (\mathcal{S}, \mathcal{A}, \Pr, R, \gamma),$$

where the solver's internal configuration $s_t = (\mathcal{A}_t, \mathcal{C}^t_{\text{working}})$ serves as the state, actions $a_t$ select the next branching variable or polarity, and transitions $\Pr(s_{t+1} \,|\, s_t, a_t)$ follow propagation and conflict resolution.

**Reward Definition.** The immediate reward measures solver progress:

$$r_t = |\mathcal{C}^{t-1}_{\text{working}}| - |\mathcal{C}^t_{\text{working}}| - \lambda \, \mathbb{I}[\texttt{conflict}],$$

where $\lambda > 0$ penalizes conflicts. This mirrors the conflict-reduction reward used in propagation and restart scheduling (see [19, Algorithms 11 and 13]).

**Policy Optimization.** The policy parameters $\phi$ are optimized via the PPO objective (defined in [19]),

$$\max_{\phi} \mathbb{E}_{\pi_\phi}\left[\sum_t \gamma^t r_t\right], \qquad \gamma \in (0,1).$$

This trains the agent to favor branching actions that quickly reduce active constraints or conflicts. The learned policy can generalize across solver instances, enabling dynamic adaptation of heuristic parameters such as decay rate $\beta$ or restart interval.

**Neural Feature Encoding.** To support policy inference, solver states are embedded using graph-based or transformer encoders. Given the clause–variable graph $\mathcal{G} = (V_{\text{var}} \cup V_{\text{cl}}, E)$, variable embeddings are obtained as

$$h_v^{(k+1)} = \phi_{\text{upd}}\Big(h_v^{(k)}, \sum_{u \in \mathcal{N}(v)} \phi_{\text{msg}}(h_u^{(k)}, e_{uv})\Big)$$

Aggregated embeddings $h_v^{(K)}$ form the feature input to $\pi_\phi$, yielding adaptive scores analogous to VSIDS/CHB but learned end-to-end.

**Integration with LCG and CDCL.** In practice, learned policies replace or modulate the scoring functions in (S2) of the LCG and CDCL (see Figures 3–4). Since only decision ordering changes, logical soundness and completeness remain intact. The policy thus serves as an intelligent, data-driven heuristic layer guiding the symbolic search.

## 5.3 Evaluation Metrics and Benchmark Results

Learning-based heuristics are commonly evaluated on standardized SAT/CSP benchmarks such as SATCOMP, the MiniZinc Challenge, and synthetic instance families. Their performance is compared to classical VSIDS and CHB baselines using several metrics: (i) **runtime**, the average CPU time required to reach SAT or UNSAT; (ii) **conflicts**, the total number of conflicts encountered before termination; (iii) **decision depth**, the average decision level at which a solution or conflict is found; (iv) **clause quality**, typically measured by the mean LBD of learned clauses; and (v) **generalization**, the solver's robustness on unseen instance distributions.

Empirical results in recent studies show that learned branching and polarity policies can reduce conflict counts by 20–40% and runtime by up to an order of magnitude on structured CSP/SAT benchmarks [21, 35, 41]. Neural policies

consistently outperform fixed-parameter heuristics in dynamic or non-stationary problem families, while maintaining solver correctness.

Heuristic search has moved beyond fixed scoring rules and now incorporates adaptive policies learned from data. By introducing learned components into variable selection and propagation, modern solvers can improve average-case performance and generalize across instance families. This connects classical `VSIDS/CHB` heuristics with reinforcement-guided decision methods.

# 6 Learning for `DisP`

`DisP` supplies a unified formulation for problems that include explicit logical alternatives among constraints. Building on the work of Balas [4], it represents either–or and one-of-many choices in a form compatible with continuous optimization, allowing logical structure to be handled within a common mathematical framework. This section revisits the foundations of `DisP` in order to clarify how logical disjunctions can be expressed as optimization objects and how their convexifications allow tractable computation. Each disjunction describes a collection of feasible regions that are locally convex but globally nonconvex, and whose union represents the combinatorial structure of many real-world decision problems. Understanding this representation prepares the ground for learning-based extensions in later sections, where neural and reinforcement learning methods can approximate or guide the resolution of these logical alternatives. In this way, `DisP` serves as a theoretical foundation for modern mixed-integer and hybrid optimization while also offering a natural platform for learning-augmented models that automatically discover useful cuts, branching rules, or feasibility patterns by combining the expressiveness of disjunctive reasoning with the adaptability of data-driven methods.

## 6.1 Technical Formulation of `DisP`

`DisP`, originally introduced by Balas [4] in the 1970s, is an optimization framework designed to handle problems with *logical disjunctions* among constraints. A (linear) disjunctive set is the solution set of systems of inequalities connected by logical operators such as conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). While each polyhedron defined by such inequalities is convex, their union is generally nonconvex, which makes disjunctive sets a natural representation of nonconvex feasible regions. The theoretical significance of `DisP` lies in the fact that it was the first broad class of nonconvex optimization problems amenable to compact convexification: the closed convex hull of a union of polyhedra can itself be represented as a polyhedron in a lifted higher-dimensional space. This result established one of the first systematic bridges between convex and nonconvex programming [4].

Moreover, `DisP` is tightly connected to integer programming. Logical conditions in real-world problems—such as either/or decisions, threshold effects, or sequencing rules—were traditionally modeled by binary variables and the "big-M" method. Conversely, integrality constraints can themselves be expressed through disjunctions. This duality enabled the development of powerful *disjunctive cuts* (e.g., intersection cuts, lift-and-project cuts) which approximate the integer hull and form the basis of modern cutting-plane methods in mixed-integer optimization. From a practical standpoint, `DisP` provides a natural modeling language for discrete choices and combinatorial structure, with applications in scheduling, routing, facility location, and configuration problems.

Formally, a mixed-integer disjunctive nonlinear program (MIDNP) can be written as

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & x \in C_{\text{DisP}}, \end{array} \tag{6}$$

where $f : C_{\text{DisP}} \subseteq \mathbb{R}^n \to \mathbb{R}$ is possibly nonconvex, and the feasible region is given by

$$C_{\text{DisP}} := \left\{ x \in \mathcal{X} \mid \bigvee_{l=1}^{L} \big(g^l(x) = 0,\ h^l(x) \le 0\big),\ x_i \in s_i\mathbb{Z} \text{ for } i \in I \right\}, \tag{7}$$

with $\mathcal{X} \subseteq \mathbb{R}^n$ a bounded domain, $I \subseteq [n]$ the set of integer variables, and $g^l, h^l$ defining the equality and inequality systems for each disjunct $l \in [L]$.

The fundamental logical disjunction governing the structure of (7) can be expressed abstractly as

$$(g^1(x) = 0,\ h^1(x) \le 0) \vee (g^2(x) = 0,\ h^2(x) \le 0) \vee \cdots \vee (g^L(x) = 0,\ h^L(x) \le 0), \tag{8}$$

representing $L$ alternative regimes of feasibility. This disjunctive formulation is the primary construct in mixed-integer `DisP` and is applied throughout the subsequent examples and algorithmic developments.

## 6.2 ML and RL Enhancements

**Differentiable Neural Logic for Disjunctive Reasoning.** This paragraph introduces the `dNL` formulation, defining continuous conjunction and disjunction operators $f_{\text{conj}}$ and $f_{\text{disj}}$[1], the initialization of logic-layer parameters ($w_i, c, x_i \in [0, 1]$), and the differentiable logical loss $L_{\text{logic}}$. These constructs enable smooth gradient-based reasoning over symbolic disjunctions. In this formulation, $x_i \in [0, 1]$ denotes the soft activation of literal $i$, interpretable as a relaxed truth

[1] $f_{\text{conj}}/f_{\text{disj}}$ are smooth min/max relaxations defined in [19, (9)]; the logical loss $L_{\text{logic}}$ is defined in [19, (10)].

value. Each literal is associated with a nonnegative weight $w_i$ that controls its contribution to the differentiable conjunction or disjunction. The temperature parameter $c > 0$ regulates the sharpness of the transition: increasing $c$ makes the operators $f_{\text{conj}}$ and $f_{\text{disj}}$ more closely approximate their Boolean counterparts, whereas smaller values yield smoother relaxations for gradient propagation.

This concept is used in steps ($S0_{17}$–$S3_{17}$) of [19, Algorithm 17] for differentiable reasoning, reused in steps ($S0_{18}$–$S3_{18}$) of [19, Algorithm 18] for neural-assisted logical updates, and referenced in steps ($S2_{11}$–$S3_{11}$) of [19, Algorithm 11] for backpropagation through logical constraints.

Algorithms 11, 17, and 18 follow a common step structure: S0 initializes the relevant data structures, S1 performs forward computation, S2 executes learning or parameter updates, and S3 integrates the updated quantities into the solver state. While these steps are indexed by algorithm (e.g., $S0_{11}$, $S0_{17}$, $S0_{18}$ in [19]), their functional roles are consistent. Algorithm 11 is originally defined for ML-enhanced propagation in `LCG`/`CDCL` solvers; here, only its learning-update phase (S2–S3) is used.

In Algorithm 17 ($S0_{17}$–$S3_{17}$), the differentiable logic layer is initialized, the operators $f_{\text{conj}}$ and $f_{\text{disj}}$ are applied, and the logical loss $L_{\text{logic}}$ is computed. In Algorithm 11 ($S2_{11}$–$S3_{11}$), gradients of $L_{\text{logic}}$ are propagated through these operators to update the parameters $(w_i, c, x_i)$ associated with clauses and literals. Algorithm 18 ($S0_{18}$–$S3_{18}$) applies the same operators to refine clause activations and variable scores before resuming symbolic reasoning.

Algorithm 18 extends this procedure to relational and grounded logical constraints by grounding relational clauses, performing unification, and incorporating neural predicate relevance scores prior to passing the resulting ground clauses to the underlying `LCG`/`CDCL` search.

The differentiable logical operators are taken from the `dNL` formulation described in [19, Section 1]. Algorithm 18 [19] summarizes their integration in learning disjunctive constraints and illustrates how differentiable neural logic embeds symbolic disjunctions into continuous optimization. Figure 9 provides a simplified flowchart (adapted from Algorithm 16) showing how differentiable logic layers enable smooth reasoning over clause activations.

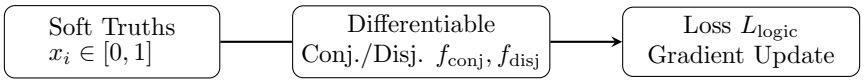



Figure 9: Differentiable neural logic (`dNL`): continuous conjunction/disjunction operators enable gradient-based optimization over logical constraints.

**RL-Guided Branching.** In addition to differentiable clause reasoning, the solver employs an RL-based branching policy that adjusts variable-selection

preferences during search. The policy observes signals derived from the differentiable logic layer, such as soft clause violations and values related to $L_{\text{logic}}$, and maps them to branching scores used by the symbolic solver.

This functionality is realized through Algorithm 11 of [19], where the initial steps ($S0_{11}$–$S1_{11}$) construct the policy state from clause- and literal-level signals, and the learning steps ($S2_{11}$–$S3_{11}$) update the policy parameters. The resulting policy modifies variable scores before each symbolic decision, thereby biasing branching toward assignments that reduce soft violations while remaining fully compatible with the underlying `SAT` search procedure.

**Neural-Assisted Clause and Variable Updates.** In addition to guiding branching decisions, the learned signals also influence clause activity and variable scoring within the symbolic engine. This mechanism is formalized in [19, Algorithm 18], whose steps ($S0_{18}$–$S3_{18}$) refine literal and clause activations using the differentiable operators introduced above. The resulting scores serve as soft heuristics that modulate clause activity, polarity preferences, and restart scheduling within the underlying `CDCL` loop.

In practice, the neural updates operate as an adaptive overlay on the classical heuristics: the learned activations bias the solver toward clauses with high soft violation scores, while still respecting all symbolic invariants. This enables a hybrid control flow in which symbolic propagation, neural feedback, and RL-guided branching jointly shape the solver's trajectory through the search space.

## 6.3 Synthesis: `DisP` Meets Learning

This section discusses how `DisP` integrates with ML and RL to combine exact logical feasibility with adaptive optimization. It establishes a theoretical basis for using neural and RL methods to approximate disjunctive systems in scheduling and planning.

The synergy between `DisP` and ML or RL lies in combining the exact logical structure of `DisP` with the adaptive, generalizing capabilities of learning algorithms. `DisP` provides a formal scaffold defining feasible regions, while ML and RL supply scalable, adaptive optimization strategies. This hybrid paradigm has shown promise in job-shop scheduling, program analysis, and production planning, and opens avenues for uncertainty-aware, real-time decision-making in complex disjunctive environments.

Detailed mathematical formulations, optimization objectives, and graphical illustrations for the representative domains listed in Table 4.

Table 4: Representative ML/RL enhancements in DisP.

| Problem Domain | ML/RL Contribution within Disjunctive Models | Reference |
|---|---|---|
| Job-shop scheduling | GNN + RL for sequential decisions on disjunctive graphs; transferable scheduling policies. | [28] |
| Job-shop scheduling | Attention-based deep RL with disjunctive graph embeddings; scalable and high-quality solutions. | [12] |
| Chemical production scheduling | Distributional RL handling precedence/disjunctive constraints; risk-sensitive optimization (CVaR). | [26] |
| Neuro-symbolic RL | Inductive logic programming integrated with RL; interpretable disjunctive decision rules. | [9] |
| Program analysis | Data-driven disjunctive modeling for selective context sensitivity; disjunction as an ML bias. | [17] |

### 6.4 Discussion

The surveyed studies report performance improvements in selected settings, especially when learned components are used as heuristic guidance while the underlying symbolic solver preserves correctness. At the same time, the evidence is heterogeneous: gains depend on the benchmark distribution, the cost of training data, the runtime overhead of the learned model, and the ability of the learned policy to transfer to unseen instance families.

## 7 Current Challenges and Future Directions

Learning–augmented constraint optimization has advanced rapidly, but several difficulties still prevent a tight integration of symbolic and statistical approaches [10, 11, 24, 29]. Present neural and RL components often do not scale well beyond standard benchmark sets and may fail to transfer to real–world or structurally different instances. These issues arise from overfitting to particular graph or clause distributions, together with the rapid combinatorial growth of the search space [10, 24].

**Interpretability and Guarantees.** In contrast to classical CDCL, LCG, or DisP frameworks, neural modules offer limited transparency and lack guarantees of soundness, completeness, or convergence. Designing explainable architectures

and developing formal analyses of learned branching and policy mechanisms remain open problems [10, 11].

**Data Efficiency.** Supervised models require large sets of solved instances, and RL methods often contend with sparse rewards and slow convergence. Self-supervised objectives, consistency-based signals, curriculum strategies, and generative instance models may reduce these data requirements, but their systematic use is still at an early stage [10, 24].

**Integration with Classical Heuristics.** Although ML-guided heuristics such as backbone prediction, learning–rate branching, and neural restart scheduling have demonstrated improvements in selected settings, their interaction with deterministic solver cores is still largely unstructured. One possible direction is to model propagation, branching, and backjumping using differentiable or stochastic operators, allowing solver parameters and learned policies to be expressed within a common optimization framework [11, 29].

**Hybrid Symbolic–Neural Solvers.** A long-term goal is to develop architectures that combine the logical guarantees of symbolic inference with the flexibility of neural representations. Existing work on differentiable logical layers and neural predicate embeddings provides early progress, but a robust and verifiable hybrid pipeline has yet to be fully developed [10, 11].

**Benchmarking and Reproducibility.** Experimental protocols for ML-enhanced solvers vary widely. Extending community benchmarks such as `SATCOMP` or the `MiniZinc` Challenge to cover learning-based systems would improve reproducibility and enable more consistent comparisons of generalization behavior [11, 29].

**Cross-Domain Integration.** Additional opportunities arise in extending learning-augmented techniques to hybrid optimization, control, and reasoning under uncertainty. Doing so may support solvers capable of addressing logical, numerical, and dynamic constraints within a unified setting [11, 24].

**Outlook.** These challenges point to several directions for future work on hybrid solvers. Advances will require scalable graph-based representations, interpretable architectures, and frameworks that relate learned components to logical inference. Addressing these problems is essential for making learning-augmented reasoning systems more reliable and widely applicable.

# 8 Conclusion

This paper presented a solver-architecture taxonomy of learning-augmented combinatorial solvers. The taxonomy organizes existing work according to the solver-level decision affected by learning: propagation, branching, conflict analysis, restart control, structural analysis, solver selection, and formulation-level reasoning. This viewpoint complements surveys organized by learning paradigm, application domain, or representation class.

The main message of the survey is that learned components are most defensible when they act as guidance mechanisms inside solver architectures whose symbolic core still preserves correctness. In this role, ML and RL models can support decisions such as variable selection, polarity prediction, restart scheduling, clause or nogood management, structural analysis, and formulation-level reasoning. These interventions can improve performance in selected settings, but their effectiveness depends on benchmark distribution, training data, runtime overhead, and transfer to unseen instance families.

The taxonomy also highlights several limitations of current learning-augmented solver designs. Neural and reinforcement-learning components often remain difficult to interpret, expensive to train, and sensitive to changes in the instance distribution. Moreover, learned guidance must be integrated carefully so that it does not compromise the soundness, completeness, or reliability of the underlying solver. Future work should therefore focus on principled interfaces between learned modules and symbolic inference, reproducible evaluation protocols, stronger generalization tests, and theoretical analyses of hybrid solver behavior.

Overall, the solver-architecture perspective helps clarify where learning can be inserted into combinatorial solvers, what information it uses, what benefit it is expected to provide, and what risks arise from the integration. This provides a structured basis for developing hybrid solvers that combine symbolic reasoning with data-driven guidance while maintaining the safeguards required for reliable combinatorial search.

**Funding** V.K. acknowledges funding support from the National Centre for Energy II (TN02000025). M.K. acknowledges support from the same project for work carried out in 2025 and, from 2026 onward, financial support from the Austrian Science Fund under https://doi.org/10.55776/PAT2747625.

**Supplementary Information** Supplementary material accompanies this article as `suppMat.pdf`. It provides mathematical background, extended pseudocode, flowcharts, examples, and empirical summaries from the literature supporting the solver-architecture taxonomy and Algorithms 1–18 discussed in the

main manuscript. An online copy is available at https://github.com/GS1400/suppMat_ML_CSP.

**Author Contributions.** MK contributed to conceptualization, methodology, literature review, taxonomy development, mathematical formulation, algorithmic organization, preparation of the main manuscript, preparation of the supplemental material, writing–original draft, and manuscript revision. VK contributed to conceptual guidance, methodological discussion, supervision, critical review of the taxonomy and manuscript structure, writing–review and editing, and manuscript revision. Both authors read and approved the final version of the manuscript.

# References


[1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3146–3153, 2020.

[2] Ravindra K Ahujia, Thomas L Magnanti, and James B Orlin. Network flows: Theory, algorithms and applications, 1993.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404, 2009.

[4] Egon Balas. *Disjunctive Programming*. Springer International Publishing, 2018.

[5] Dina Barak-Pelleg and Daniel Berend. On the satisfiability threshold of random community-structured sat. pages 1249–1255, 2018.

[6] Ruth Helen Bergin, Marco Dalla, Andrea Visentin, Barry O'Sullivan, and Gregory Provan. Using machine learning classifiers in sat branching. In *Proceedings of the International Symposium on Combinatorial Search*, volume 16, pages 169–170, 2023.

[7] Filip Beskyd and Pavel Surynek. Parameter setting in sat solver using machine learning techniques. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence*, page 586–597. SCITEPRESS - Science and Technology Publications, 2022.

[8] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In Tomáš Balyo, Matti Järvisalo, and Markus Iser, editors, *Proceedings of SAT Competition 2020 – Solver and Benchmark*

*Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 4–12, Helsinki, Finland, 2020. University of Helsinki.

[9] Andreas Bueff and Vaishak Belle. Deep inductive logic programming meets reinforcement learning. In *Proceedings of the 39th International Conference on Logic Programming (ICLP 2023)*, volume 385 of *EPTCS*, pages 339–352, 2023.

[10] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

[11] Quentin Cappart, Tias Guns, Michele Lombardi, Gilles Pesant, and Dimos Tsouros. Combining constraint programming and machine learning: From current progress to future opportunities. *Journal of Artificial Intelligence Research*, 84:1–52, 2025.

[12] Ruiqi Chen, Wenxin Li, and Hongbing Yang. A deep reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for the job-shop scheduling problem. *IEEE Transactions on Industrial Informatics*, 19(2):1322–1334, 2023.

[13] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 265–279. Springer, 2002.

[14] Márk Danisovszky, Zijian Gyozo Yang, and Gábor Kusper. Classification of sat problem instances by machine learning methods. In *ICAI*, pages 94–104, 2020.

[15] Emir Demirović, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J. Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26):1–47, 2022.

[16] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[17] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. A machine-learning algorithm with disjunctive model for data-driven program analysis. *ACM Transactions on Programming Languages and Systems*, 41(2):13:1–13:41, 2019.

[18] Morteza Kimiaei and Vyacheslav Kungurtsev. Learning-Augmented Combinatorial Solvers: A Solver-Architecture Taxonomy. Optimization Online E-Print Repository, November 2025. Published: 2025-11-09; updated: 2026-10-05; ID: 32520.

[19] Morteza Kimiaei and Vyacheslav Kungurtsev. Supplemental material for *Learning-Augmented Combinatorial Solvers: A Solver-Architecture Taxonomy*. https://github.com/GS1400/suppMat_ML_CSP, 2026. Supplemental material; accessed: 2026-10-05.

[20] Jia Hui Liang. *Machine Learning for SAT Solvers*. Phd thesis, University of Waterloo, 2018.

[21] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer, 2016.

[22] Chien-Liang Liu and Tzu-Hsuan Huang. Dynamic job-shop scheduling problems using graph neural network and deep reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 53(11):6836–6849, 2023.

[23] Gonçalo P. Matos, Luís M. Albino, Ricardo L. Saldanha, and Ernesto M. Morgado. Solving periodic timetabling problems with sat and machine learning. *Public Transport*, 13(3):625–648, 2021.

[24] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

[25] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[26] Max Mowbray, Dongda Zhang, and Ehecatl Antonio Del Rio Chanona. Distributional reinforcement learning for scheduling of chemical production processes. 2022.

[27] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. Learning optimal decision trees with sat. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1362–1368. IJCAI Organization, 2018.

[28] Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11):3360–3377, 2021.

[29] Andrei Popescu, Seda Polat-Erdeniz, Alexander Felfernig, Mathias Uta, Müslüm Atas, Viet-Man Le, Klaus Pilsl, Martin Enzelsberger, and Thi Ngoc Trang Tran. An overview of machine learning techniques in constraint solving. *Journal of Intelligent Information Systems*, 58(1):91–118, 2022.

[30] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[31] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, Harlow, England, 4th, global edition edition, 2021.

[32] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode: Generic constraint development environment. https://www.gecode.org, 2006. Accessed: 2025-11-04.

[33] Daniel Selsam and Nikolaj Bjørner. *Guiding High-Performance SAT Solvers with Unsat-Core Predictions*, page 336–353. Springer International Publishing, 2019.

[34] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[35] Feng Shi, Chonghan Lee, Mohammad Khairul Bashar, Nikhil Shukla, Song-Chun Zhu, and Vijaykrishnan Narayanan. Transformer-based machine learning for fast sat solvers and logic synthesis. *arXiv preprint arXiv:2107.07116*, 2021.

[36] Zhengyuan Shi, Min Li, Sadaf Khan, Hui-Ling Zhen, Mingxuan Yuan, and Qiang Xu. Satformer: Transformers for sat solving. *arXiv preprint arXiv:2209.00953*, 2022.

[37] Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, Kathryn Francis, and Geoffrey Chu. Chuffed: A lazy clause generation solver. https://github.com/chuffed/chuffed, 2018. Accessed: 2025-11-04.

[38] Ling Sun, David Gerault, Adrien Benamira, and Thomas Peyrin. Neurogift: Using a machine learning based sat solver for cryptanalysis. In *Proceedings of the 4th International Symposium on Cyber Security Cryptography and Machine Learning (CSCML 2020)*, volume 12161 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2020.

[39] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[40] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. *Constraints*, 25(3–4):226–250, 2020.

[41] Haoze Wu. Improving sat-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 787–788. ACM, March 2017.

## Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- suppMat.pdf