Machine Learning Algorithms for Assisting Solvers for Constraint Satisfaction Problems

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 survey proposes a unifying conceptual framework and taxonomy that systematically integrates Machine Learning (ML) and Reinforcement Learning (RL) with classical paradigms for Constraint Satisfaction and Boolean Satisfiability solving. Unlike prior reviews that focus on individual applications, we organize the literature around solver architecture, linking each major phase—constraint propagation, heuristic decision-making, conflict analysis, and meta-level structural learning—to its corresponding learning paradigm. We review the evolution from symbolic constraint propagation to modern neuro-symbolic optimization, highlighting the methodological convergence between Operations Research and Artificial Intelligence. Building upon the Lazy Clause Generation and Conflict-Driven Clause Learning architectures, we introduce ML/RL-enhanced algorithmic modules that demonstrate how data-driven inference can augment logical reasoning. Our proposed taxonomy connects solver components to specific learning approaches, including Graph Neural Networks, Transformer encoders, and policy-gradient algorithms such as Proximal Policy Optimization. We identify key research challenges—particularly the preservation of logical soundness, generalization across problem distributions, and interpretability of learned heuristics—and outline a roadmap toward scalable hybrid optimization frameworks that unify symbolic reasoning with data-driven learning.

Keywords.

Constraint Satisfaction Problem; Boolean Satisfiability; Lazy Clause Generation; Conflict-Driven Clause Learning; Machine Learning; Reinforcement Learning; Neuro-Symbolic Optimization; Graph Neural Networks; Transformer Models; Learned Heuristics; Hybrid Solvers.

Contents

1 Introduction

Optimization and constraint reasoning lie at the foundation of both Operations Research (OR) and Artificial Intelligence (AI). Constraint Satisfaction Problems (`CSPs`) and Boolean Satisfiability (`SAT`) formulations provide the mathematical core for decision-making under combinatorial and logical constraints. They underpin a wide range of industrial applications, including scheduling, routing, planning, and verification. Classical solver architectures such as `LCG` (Lazy Clause Generation) and `CDCL` (Conflict-Driven Clause Learning) [5,13,21,47,53] deliver provably correct results but face scalability bottlenecks as problem size and complexity grow. In parallel, Machine Learning (ML) and Reinforcement Learning (RL) have emerged as powerful paradigms for capturing structure, predicting solver behavior, and guiding search heuristics in data–driven ways. Neural Networks (NNs), and particularly Graph Neural Networks (GNNs), have become central tools in this context, providing structured representations of variable–clause interactions and enabling solvers to learn from the graph topology of constraint systems. The convergence of these fields has sparked the development of *learning-augmented solvers*, which combine symbolic reasoning with statistical inference to address large-scale and dynamic constraint satisfaction.

1.1 Problem Setting

This survey investigates the intersection of classical constraint optimization and learning–based methodologies. Specifically, it examines how ML and RL techniques can be embedded within `CSP` and `SAT` solvers to improve propagation, branching, conflict analysis, and restart strategies. Unlike empirical benchmarks or algorithmic proposals, this work aims to consolidate and analyze the growing body of research that integrates data–driven models into solver pipelines. Formally, let a `CSP` instance be defined as $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ with variables $\mathcal{X}$, domains $\mathcal{D}$, and constraints $\mathcal{C}$. Traditional solvers explore the feasible assignment space via deterministic search; learning-augmented solvers instead estimate probabilistic guidance functions π_θ or predictive mappings f_θ that influence these steps. This survey therefore focuses on the systematic mapping between classical solver phases and their ML/RL-enhanced counterparts.

1.2 Challenges in Constraint Satisfaction

Despite rapid progress, several persistent challenges motivate this survey:

- **Scalability.** Classical `CSP`/`SAT` solvers face exponential search spaces, while ML models struggle to generalize to unseen instance structures.

- **Interpretability.** Neural solvers lack the logical transparency of rule-based algorithms, hindering verification and trust in critical domains.
- **Data and Transferability.** Collecting labeled solver traces or optimal solutions is expensive, limiting supervised learning and domain transfer.
- **Integration Consistency.** Many hybrid architectures remain ad hoc, lacking theoretical frameworks to ensure completeness or soundness when learning modules intervene in logical reasoning.

These issues define the research landscape that this survey seeks to organize and clarify.

1.3 Research Questions

This survey is structured around three guiding research questions that jointly define how learning paradigms intersect with classical 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. How have ML and RL been integrated into the main phases of `CSP/SAT` solvers? This question examines learning-based enhancements to propagation, decision-making, conflict analysis, and restart strategies within `LCG/CDCL` frameworks.

Q2. What algorithmic and representational frameworks support this integration? It investigates the role of GNN-based embeddings, differentiable logic, reinforcement-guided heuristic adaptation, and other hybrid architectures that enable data-driven reasoning.

Q3. What are the principal open challenges and research trends? This includes the issues of scalability, interpretability, data efficiency, and theoretical soundness that define the frontier of learning–augmented reasoning.

Together, these questions structure the taxonomy, analysis, and synthesis presented throughout the paper, linking foundational solver theory with contemporary learning-based innovations.

1.4 Survey Scope and Organization

This paper is organized as both a *survey* and a *conceptual synthesis* that bridges the traditions of OR, Constraint Programming, and AI with modern learning–based reasoning. The survey integrates foundational optimization frameworks [3,6,29] with contemporary advances in ML, RL, and graph neural architectures [22,55,57,60].

Section 2 introduces the mathematical foundations of `CSPs` and `SAT`, revisiting core solver architectures such as `CDCL` and `LCG` [5,13,21,47,53]. Section 3 reviews background concepts from ML and RL that are central to solver guidance, branching, and parameter optimization [11,32,36,46].

Section 4 discusses network-flow and graph-based optimization models [3,14,41], bridging these classical representations to neural embeddings and message-passing frameworks [22,24].

Sections 5 and 6 analyze direct learning and heuristic search strategies for traditional solvers, while Sections 7 and 8 explore reinforcement-guided heuristics and GNNs for structured reasoning [50,51,57,60].

Section 9 extends these ideas to `DisP` and hybrid scheduling formulations [6,16,34,42], highlighting neural and RL-based dispatching policies [40]. Then, it generalizes these models to higher-order and first-order logical `CSPs`, emphasizing neuro-symbolic reasoning and differentiable logic frameworks [15,43].

Section 10 consolidates empirical results from recent studies demonstrating the effectiveness of ML- and RL-assisted constraint solving and combinatorial optimization. It shows that data-driven heuristics and hybrid neural–symbolic frameworks consistently enhance solver scalability, runtime efficiency, and solution quality across diverse domains.

Section 11 synthesizes the open challenges of scalability, interpretability, and theoretical soundness, while Section 12 presents future research directions for learning-augmented reasoning systems. Across all sections, the survey maintains a unifying perspective grounded in the integration of optimization theory, constraint solving, and data–driven inference, as illustrated in Figure 1.

1.5 Our Contributions

As a survey and conceptual synthesis, this work **proposes a unifying taxonomy and conceptual framework** that systematically integrates ML and RL within classical `CSP` and `SAT` solving architectures. Unlike prior reviews that describe isolated applications, we organize the field around the internal phases of modern solvers, showing how each component—propagation, branching, conflict analysis, meta-level reasoning, and integration—can be enhanced

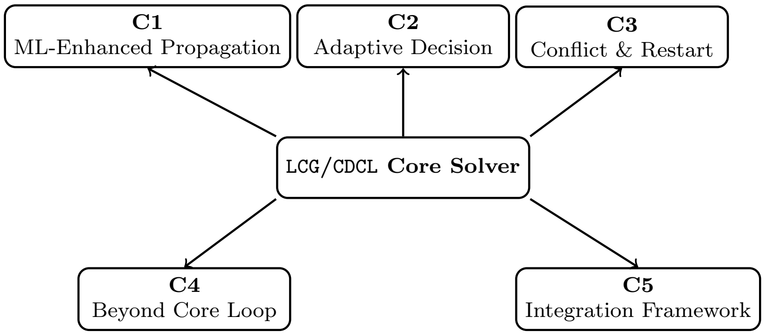


Figure 1: Compact overview of the unified ML/RL–CSP/SAT framework highlighting five key contribution areas.

by learning-based methods. Building upon the foundational LCG (Lazy Clause Generation) and CDCL (Conflict-Driven Clause Learning) paradigms, the framework identifies five core areas where learning techniques have been successfully or potentially embedded within solver pipelines. These contributions are summarized in Figure 1 and detailed below.

C1. ML-Enhanced Propagation. A propagation framework that integrates logistic regression and Monte Carlo backbone prediction with RL-based search-space pruning. This module (Algorithm 3) replaces static domain filtering with data–driven backbone estimation, thereby improving consistency enforcement and reducing conflict depth. The method builds upon prior work on backbone prediction in MiniSAT [59] and RL-based constraint pruning in periodic timetabling [35], and extends the abstraction–refinement approach introduced in [17].

C2. Adaptive Heuristic Decision. A learned decision process (Algorithm 4) combining instance classification, parameter optimization, and neural branching. The approach generalizes the Learning Rate Branching (LRB) heuristic [32] and integrates transformer-based guidance [50] and classifier-driven branching [10]. Parameter tuning builds on the framework proposed in [11] and polarity prediction models from [59]. Together, these techniques yield adaptive solver configurations that dynamically align with problem structure.

C3. ML/RL-Guided Conflict Analysis and Backjumping. A conflict management model (Algorithm 5) that predicts clause utility using regression or neural estimators and applies RL to optimize restart policies. The design draws on the RL-inspired restart scheduling of [32], the domain-driven RL policies for scheduling problems in [35], and the neural prioritization mechanisms developed in NeuroGIFT for cryptanalysis [54]. Specifically, NeuroGIFT applies neural ranking to candidate assignments in cryptanalytic SAT formulations rather than general clause learning, illustrating how learned prioritization can enhance structured search in specialized domains. This integration enhances clause retention efficiency, restart scheduling, and solver convergence on structured instances.

C4. Learning Beyond the Core Loop. A meta-level module (Algorithm 6) that applies ML to study community structure and phase transitions in SAT instances [8], and leverages SAT-encoded learning for interpretable models such as optimal decision trees [39]. This unifies structural analysis and symbolic learning, contributing to explainable and hybrid neuro-symbolic optimization.

C5. Comprehensive Integration Framework. The paper synthesizes these methods into a consistent taxonomy linking LCG/CDCL steps to ML and RL enhancements. Table 5, below, summarizes this integration, providing a blueprint for developing hybrid intelligent solvers that combine statistical inference and logical reasoning [8,32,39,50,59].

Future research should further formalize the theoretical guarantees of this unified architecture and explore its scalability to real-world, large-scale constraint domains.

2 Classical Foundations of CSPs

CSPs provide a unifying mathematical formalism for expressing a wide range of discrete decision-making tasks that lie at the intersection of OR and AI. They represent problems in which feasible solutions must satisfy a collection of logical, algebraic, or combinatorial constraints, encompassing models such as SAT, graph coloring, scheduling, and network design. Over the past decades, the study of CSPs has not only shaped the theoretical foundations of computational complexity and combinatorial optimization but has also guided the design of practical algorithms based on propagation, search, and logical inference.

Goal and intuition. This section revisits the classical formulations and algorithmic paradigms underlying CSPs to clarify how constraint representation, propagation, and search interact to define feasible reasoning. The intuition is to expose the logical and combinatorial structures that later serve as "anchors" for ML and RL enhancements.

Motivation. By grounding subsequent developments in these foundations, we highlight why classical CSP techniques remain central: they offer a principled framework into which learning components can be systematically integrated, ensuring that scalability and adaptability are achieved without sacrificing logical soundness.

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.

Formally, a CSP can be defined as a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where

- $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ is a finite set of decision variables,
- $\mathcal{D} = \{\mathcal{D}(x_1), \mathcal{D}(x_2), \dots, \mathcal{D}(x_n)\}$ 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),
- $\mathcal{C} = \{c_1, c_2, \dots, c_q\}$ is the set of constraints. 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 $\{S_{i1}, S_{i2}, \dots, S_{iN_i}\}$, i.e.

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

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: logical (propositional or first-order) formulas can be directly translated into 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 the **first-order logic (FOL)** setting, constraints extend to quantified predicates over structured domains, for example

$$\forall x_i, x_j \; [\, R(x_i, x_j) \Rightarrow \neg C(x_i, x_j) \,],$$

which expresses that relation R forbids simultaneous satisfaction of condition C. Both cases fit naturally within the general CSP framework $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, allowing logical and numerical constraints to coexist under a unified representation.

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:

- **Finite CSPs:** all domains $\mathcal{D}(x_i)$ are finite;
- **Infinite CSPs:** at least one $\mathcal{D}(x_i)$ is infinite;
- **Boolean CSPs:** all domains $\mathcal{D}(x_i) = \{0, 1\}$;
- **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

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

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

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

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

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

Here, $A \in \mathbb{R}^{n\times m}$, $B \in \mathbb{R}^{n\times p}$, $b \in \mathbb{R}^{m}$, $d \in \mathbb{R}^{p}$, and $c \in \mathbb{R}^{n}$. Structural combinatorial optimization focuses on exploiting properties of C_{int} or C_{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 (COP)*:

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

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 (MRV) 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` [44, 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 search algorithms such as depth-first search (DFS), breadth-first search (BFS), and branch-and-bound (BB) form the algorithmic backbone of constraint satisfaction problem (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 Appendix A.1.

Table 1 summarizes the asymptotic time and space complexities of the six foundational algorithms discussed in this chapter. As described by [44, Ch. 5.2], the constraint-propagation algorithm AC-3 runs in $O(c|\mathcal{D}|^3)$ time and $O(c|\mathcal{D}|^2)$ space, providing efficient domain pruning for binary constraints. Search-based procedures such as BTS, BFS, and DFS exhibit exponential worst-case complexity [44, Chs. 3.4–3.6, 5.3], reflecting the NP-completeness of general CSPs. BB inherits the exponential worst-case growth of DFS, but often achieves substantial practical gains through bounding and pruning strategies [6, 44, Ch. 4]. The informed search algorithm A* maintains the same exponential bound in the worst case, but can achieve dramatic speedups when using consistent heuristics.

These asymptotic limits underscore why subsequent sections introduce heuristic, learning-based, and neural approaches. By guiding search with data-driven

Table 1: Asymptotic time and space complexities of the six classical algorithms discussed in this section, based on [6,44]. Here, b is the branching factor, d the depth of the shallowest solution, m the maximum search depth, n the number of variables, c the number of constraints, and $|\mathcal{D}|$ the maximum domain size.

Algorithm	**Time Complexity**	**Space Complexity**
`AC-3`	$O(c\,\lvert\mathcal{D}\rvert^3)$	$O(c\,\lvert\mathcal{D}\rvert^2)$
`BTS`	$O(\lvert\mathcal{D}\rvert^n)$ (worst case)	$O(n)$
`BB`	exponential in n	$O(n)$
`BFS`	$O(b^{d+1})$	$O(b^{d+1})$
`DFS`	$O(b^m)$	$O(bm)$
`A*`	$O(b^d)$ (worst case)	$O(b^d)$

inference, such methods aim to mitigate exponential growth and improve the scalability of solver decisions.

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 [6], 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 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 algorithmic connection highlights how logical inference and convex analysis can jointly enforce feasibility in hybrid discrete–continuous systems.

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 are frequently used as valid cuts in scheduling and sequencing formulations, bridging the logical and geometric representations of precedence.

In hybrid learning–based solvers, disjunctive constraints are embedded into neural propagation modules that estimate clause activation probabilities. Given a learned clause embedding $\phi_k = \mathrm{Enc}_\theta(A^{(k)}, b^{(k)})$, 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, Enc_θ denotes a neural encoder (defined by (14) in Appendix B.1).

This geometric–logical unification provided by `DisP` forms a natural bridge to the graph-structured and neural optimization frameworks developed in the subsequent sections.

3 ML, RL, and Deep NNs Background

ML and RL provide the statistical and decision-theoretic foundations for integrating adaptive intelligence into classical optimization frameworks such as `DisP` and `SAT`. Traditional solvers rely on fixed heuristics and symbolic inference, whereas ML and RL introduce mechanisms for learning from data and interaction, enabling systems to infer patterns, predict outcomes, and optimize decisions dynamically.

Goal and intuition. The goal of this section is to establish the conceptual and mathematical underpinnings that allow learning components to operate within constraint solvers. The intuition is to present ML and RL not as external tools, but as complementary reasoning mechanisms—statistical extensions of symbolic logic capable of guiding search, inference, and optimization in uncertain or data-rich settings.

Motivation. By reviewing the essential learning principles and neural architectures here, we prepare the reader to understand how these models are embedded in later sections—where they control propagation, decision making, and constraint satisfaction within hybrid symbolic–learning solvers.

3.1 ML Foundations

ML formalizes learning from data as an optimization problem over a hypothesis space of functions. Given a dataset $\mathcal{D}_{\text{train}} = \{(x_i, y_i)\}$ drawn from an unknown distribution, the learner seeks a function f_θ that minimizes the expected loss $\mathbb{E}[\ell(f_\theta(x), y)]$. Depending on the form of supervision, ML encompasses:

- *Supervised learning*, where labeled data (x_i, y_i) guide prediction (classification or regression);
- *Unsupervised learning*, which infers latent structure or representations without explicit targets (e.g., clustering, autoencoding);
- *Self-supervised or contrastive learning*, which generates supervisory signals directly from the data distribution.

Generalization—the ability to perform well on unseen instances—is achieved through regularization, inductive biases, and architectural constraints. These concepts underpin the predictive components used in learning-augmented solvers.

Detailed formulations of message-passing, attention, and regression models are given in Appendix B.1.

3.2 RL Principles

RL extends ML to sequential decision-making under uncertainty. An agent interacts with an environment modeled as a Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, \Pr, R, \gamma)$, observing states s_t, taking actions a_t, and receiving rewards r_t. The objective is to learn a policy $\pi_\theta(a|s)$ that maximizes the expected discounted return $J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_t \gamma^t r_t]$. Learning proceeds through experience, balancing exploration of new actions and exploitation of known good ones. Two complementary families of methods are widely used:

- *Value-based approaches*, which estimate optimal value functions $Q^*(s, a)$ (e.g., Q-learning, DQN);

- *Policy-gradient and actor–critic approaches*, which directly optimize the policy parameters through gradient ascent.

RL provides the algorithmic foundation for adaptive branching, scheduling, and configuration strategies introduced later in the paper, with full mathematical derivations and algorithmic specifications of policy-gradient, actor–critic, and risk-sensitive methods presented in Appendix B.1.

3.3 Technical Foundations of Deep NNs

Deep Neural Networks (DNNs) provide the parametric foundation for modern ML models. They generalize classical function-approximation schemes by composing multiple layers of nonlinear transformations that progressively extract hierarchical features from data. A standard feedforward network defines a differentiable mapping

$$f_\theta : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}, \qquad f_\theta(x) = \phi_L \circ \phi_{L-1} \circ \cdots \circ \phi_1(x),$$

where each layer ϕ_ℓ is parameterized by a weight matrix W_ℓ and bias vector b_ℓ, and applies an elementwise nonlinearity σ:

$$\phi_\ell(x) = \sigma(W_\ell x + b_\ell), \qquad \sigma(z) \in \{\text{ReLU}(z),\ \tanh(z),\ \text{sigmoid}(z)\}.$$

The parameter set $\theta = \{W_\ell, b_\ell\}_{\ell=1}^{L}$ is learned from data by minimizing an empirical loss

$$L(\theta) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x_i, y_i) \in \mathcal{D}_{\text{train}}} \ell(f_\theta(x_i), y_i),$$

where $\ell(\cdot, \cdot)$ measures prediction discrepancy (e.g., cross-entropy or mean-squared error). Parameter updates follow gradient-based optimization such as stochastic gradient descent (`SGD`) or adaptive methods like `Adam` [27]:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta L(\theta),$$

where $\eta > 0$ denotes the learning rate.

3.4 Brief Primer on DNNs

Representation Learning. Each hidden layer learns intermediate features that transform raw inputs into more abstract, task-relevant representations. Lower layers typically encode local or syntactic patterns (e.g., edge or clause statistics), while deeper layers capture global semantic or relational structure. The universal-approximation theorem guarantees that sufficiently wide DNNs can approximate any continuous mapping on a compact domain, providing a theoretical justification for their expressive power.

Regularization and Generalization. To prevent overfitting, several mechanisms are commonly used: weight-decay regularization ($\lambda\|\theta\|_2^2$), dropout (stochastic node omission), and batch normalization (adaptive rescaling of activations). These techniques improve generalization across problem instances $\phi \sim \mathcal{D}_{\text{inst}}$.

Architectural Variants. Beyond fully connected (dense) networks, specialized architectures exploit problem structure:

- **Convolutional Neural Networks (CNNs)** share weights across local neighborhoods, supporting translation invariance.
- **Recurrent Neural Networks (RNNs)** and gated variants (LSTM, GRU) capture temporal or sequential dependencies through recurrent state updates.
- **GNNs** extend these principles to relational structures (see Appendix B.1), enabling message passing over variable–constraint graphs in `CSP` and `SAT` formulations.

Backpropagation in Learning-Augmented Solvers. In neural-augmented solvers, the policy parameters θ are optimized via stochastic gradient descent:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta \mathcal{L}(\pi_\theta(\mathcal{A}_p), y),$$

where $\mathcal{L}$ measures the discrepancy between the learned policy π_θ and a target decision y. This continuous update mechanism complements the discrete backtracking of classical solvers, linking symbolic search with differentiable learning.

For detailed mathematical derivations of gradient propagation, graph-neural architectures, and reinforcement-learning algorithms, see Appendix B.1, which provides the formal training equations and algorithmic specifications underpinning these models. ML, RL, and neural architectures together offer complementary mechanisms for adaptive optimization—ML captures statistical regularities from data, RL enables sequential decision adaptation, and DNNs provide expressive parametric representations.

Integration and Notation. ML components act as predictive heuristics that refine fixed solver rules (e.g., variable scoring or branching order), while RL agents learn adaptive control policies through sequential interaction. Table 2 summarizes the notation used throughout this work.

Table 2: Summary of notation for ML and RL components used in this work.

Symbol	**Meaning**
f_θ	parameterized machine learning model.
$\pi_\phi(a\|s)$	Policy (actor) parameterized by ϕ (sometimes denoted π_θ in `PPO`/`A2C` formulations).
$V_\phi(s)$	State–value function (critic).
$Q_\psi(s,a)$	Action–value function.
$\hat{A}_t = R_t - V_\phi(s_t)$	Advantage estimate measuring deviation from baseline performance.
ρ_t	`PPO` probability ratio $\pi_\theta(a_t\|s_t)/\pi_{\theta_{\text{old}}}(a_t\|s_t)$.
ϵ	`PPO` clipping constant controlling update bounds.
$\alpha_\theta, \alpha_\phi$	Learning rates for actor and critic networks.
$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \Pr, R, \gamma)$	Markov Decision Process (MDP) tuple: states, actions, transitions, rewards, and discount factor.
s_t, a_t, r_t	State, action, and reward at time step t.
T	Rollout length or time horizon.
h_v	Learned node embedding of vertex v in a GNN.
$\mathcal{N}(v)$	Neighborhood of node v.
$\mathbf{W}$	Learnable weight matrix used in message passing.
r_t	Stochastic reward signal in bandit and RL settings.
$\boldsymbol{\theta}$	Tunable solver or model configuration parameters.
$\phi_{\text{msg}}, \phi_{\text{upd}}$	Message and update functions in GNNs.
ϕ_{read}	Readout operator aggregating node embeddings.
c	Temperature coefficient controlling sigmoid smoothness in `dNL`.
`node2vec`	Random-walk–based node-embedding method producing input features x_i for graph models.
`pointer attention`	Attention mechanism used in sequence decoders to focus on selected elements during output generation [58].
$\mathcal{C}_{\text{working}}$	Working clause set maintained during search in `SAT` solvers.
`trial`	Record of assigned literals and their corresponding decision levels.
`LBD`	Literal Block Distance, a metric for clause activity in conflict analysis.
$\boldsymbol{\eta}$	Solver or hyperparameter configuration vector (distinct from $\boldsymbol{\theta}$).

4 Network Flow Models and Applications

Network flow theory forms one of the foundational pillars of operations research (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 [3].

Goal and intuition. The goal of this section is to present the mathematical structure and intuition behind classical network flow models that underpin many combinatorial optimization problems. At its core, a network flow model translates global allocation or routing decisions into local conservation laws—capturing how limited resources can be optimally distributed across a network subject to physical or logical constraints. Understanding these models provides a geometric and algebraic foundation for later learning-based methods that operate on graph-structured data.

Motivation. By revisiting network flow formulations, we clarify how fundamental flow constraints mirror the message-passing and propagation mechanisms later used in ML- and RL-enhanced solvers. These models thus serve both as analytical benchmarks for optimization and as conceptual precursors to modern GNNs and RL frameworks that approximate or generalize them.

The detailed formulations of canonical network 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 Appendix C.1. Each model is presented as a linear or mixed-integer program illustrating the flow conservation, capacity, and logical disjunction principles that underpin 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 highlights the structural relationships among the core problem classes—from continuous flow-based formulations (e.g., traffic routing, spectrum allocation, and energy-aware design) to combinatorial disjunctive models (e.g., transport route selection and machine configuration)—that jointly form the foundation for the hybrid ML– and RL–enhanced frameworks developed later in this 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.

As defined in the introduction, `DynP` denotes dynamic disjunctive programming, i.e., time-dependent `DisP` models augmented with adaptive control. The unified framework developed in this section highlights how planning and schedul-

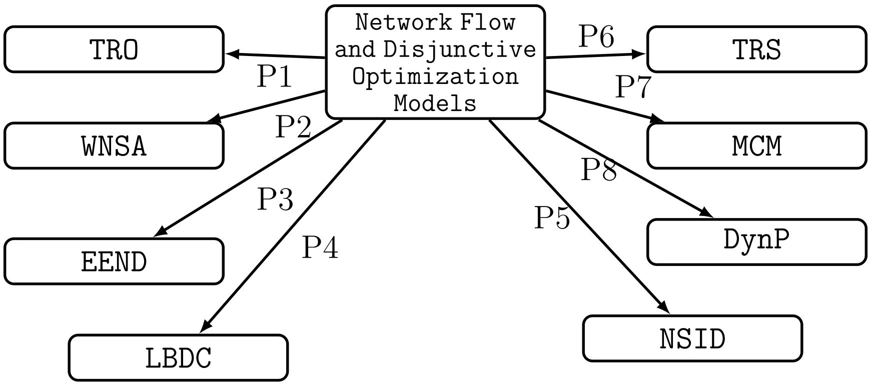


Figure 2: Overview of classical Network Flow and Disjunctive Optimization Models (P1–P8). TRO–Traffic Routing, WNSA–Wireless Spectrum Allocation, EEND–Energy-Efficient Network Design, LBDC–Load Balancing in Data Centers, NSID–Network Security and Intrusion Detection, TRS–Transport Route Selection, MCM–Machine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

ing problems can be formulated as dynamic disjunctive optimization processes. 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 3 summarizes representative learning–enhanced approaches that combine disjunctive logic, reinforcement learning, and neural inference, highlighting the diversity of architectures and optimization objectives used in practice.

Table 3: 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 $\pi_\phi(a_t \vert s_t)$ for makespan reduction	[40]
Attention–Based Deep RL	Transformer encoder captures long-range dependencies; pointer network decoder generates schedules sequentially	[16]
Dynamic GNN–RL	Online adaptation to evolving graphs $\mathcal{G}_t$ with stochastic job arrivals and breakdowns; continual policy updates via PPO	[34]
Distributional RL (Risk-Aware)	Incorporates stochastic return distributions and CVaR-based objectives for robust scheduling decisions	[38]
Differentiable Logic Models	Neural inference of logical constraints and clause activations within disjunctive or ILP formulations	[15]

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.

5 Direct Solution Learning for CSPs

Direct solution learning aims to construct models that can infer satisfying assignments for CSPs without relying on explicit symbolic search or iterative propagation. Instead of exploring the search tree step by step, these models learn to approximate the solution operator—a mapping from problem instances to feasible solutions—through data-driven generalization.

Goal and intuition. The goal of this section is to formalize how learning algorithms can replace the traditional search process with direct prediction. The central intuition is that by exposing the model to many solved instances, it can internalize the structural regularities that govern feasible assignments and thus infer new solutions in a single forward pass. This viewpoint reframes constraint solving as a supervised or self-supervised learning task: rather than discovering a solution through combinatorial reasoning, the system learns the function that generates it.

Motivation. Understanding direct solution learning is crucial because it represents the most aggressive form of learning integration into CSPs: the solver itself becomes a trained model. Such models offer a blueprint for amortized inference, neural decoding, and hybrid solvers that couple symbolic consistency checks with fast parametric prediction.

5.1 Technical Formulation of the Solution Operator

These models approximate the *solution operator*

$$\Phi : \mathcal{I} \to \mathcal{S}, \qquad \Phi(I) = \mathcal{A}_I \text{ such that } \mathcal{A}_I \models \mathcal{C}_{\text{working}}(I),$$

where $\mathcal{I}$ denotes the space of encoded CSP instances, $\mathcal{S}$ the space of feasible assignments, and $\mathcal{C}_{\text{working}}$ the active constraint set during inference. Here, $\mathcal{A}_I : \mathcal{X} \to \mathcal{D}$ is a predicted assignment that satisfies the active constraint set $\mathcal{C}_{\text{working}}(I)$. This formulation parallels the satisfiability operator used in classical search (Section 6), but replaces discrete branching with parametric inference.

5.2 Supervised and Self-Supervised Learning for CSP

Given a dataset

$$\mathcal{D}_{\text{CSP}} = \{(I_i, \mathcal{A}_i)\}_{i=1}^{N},$$

each I_i is an encoded instance, and $\mathcal{A}_i$ is a known feasible assignment satisfying all constraints in $\mathcal{C}_{\text{working}}(I_i)$. A parametric model $f_\theta : \mathcal{I} \to \mathcal{S}$ is trained to minimize

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(I_i),\, \mathcal{A}_i) + \lambda \sum_{C \in \mathcal{C}_{\text{working}}(I_i)} \mathbb{I}\big[\neg(\mathcal{A}_i \models C)\big],$$

where $\ell(\cdot,\cdot)$ measures assignment distance (e.g., mean-squared error or cross-entropy), $\lambda > 0$ penalizes constraint violations, and $\mathbb{I}$ is the indicator function. The model's prediction $\hat{\mathcal{A}}_i = f_\theta(I_i)$ approximates the satisfying assignment for instance I_i.

Self-Supervised Constraint Optimization. When labeled assignments $\mathcal{A}_i$ are unavailable, training uses constraint-driven self-supervision. Each constraint $C_j(x_{S_j})$ defines a relaxable satisfaction function $\psi_j(x_{S_j}) \in [0,1]$. The objective maximizes expected satisfaction:

$$\max_\theta \mathbb{E}_{I\sim\mathcal{D}_{\text{unsup}}}\left[\frac{1}{|\mathcal{C}_{\text{working}}(I)|}\sum_{C_j\in\mathcal{C}_{\text{working}}(I)}\psi_j(f_\theta(I)_{S_j})\right].$$

This constraint-aware objective aligns with the self-supervised principles used for backbone and polarity estimation (cf. Section 6.2).

To ensure discrete feasibility, a differentiable binarization layer $\rho_\tau(z) = \text{sigmoid}(z/\tau)$ with temperature $\tau > 0$ enforces near-binary variable values:

$$\hat{x}_i = \rho_\tau(z_i), \qquad z_i \in \mathbb{R},$$

allowing end-to-end gradient flow while approximating integral assignments.

Graph Encoding and GNN Message Passing. Each CSP instance is represented by a bipartite graph $G = (V_X \cup V_C, E)$, connecting variable nodes V_X and constraint nodes V_C. The GNN update rule (consistent with Eq. (11), defined in Appendix B.1) is

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

where $\mathcal{N}(v)$ is the neighborhood of v and ϕ_{upd}, ϕ_{msg} are neural update functions. Final embeddings $\{h_v^{(K)}\}$ are decoded by g_θ to produce variable assignments $\hat{x}_i$. This representation mirrors the clause–literal graphs used in SAT-based neural solvers (Section 6).

5.3 End-to-End Neural Solvers

End-to-end neural solvers embed both variable representation and constraint enforcement into a differentiable graph. Given an instance $I = (\mathcal{X}, \mathcal{C}_{\text{working}})$,

$$z = \text{Enc}_\theta(I), \qquad \hat{\mathcal{A}} = \text{Dec}_\phi(z),$$

where Enc_θ is a GNN or Transformer encoder, and Dec_ϕ predicts assignments $\hat{\mathcal{A}} = (\hat{x}_1, \ldots, \hat{x}_n)$. Constraint satisfaction is measured by

$$R(I, \hat{\mathcal{A}}) = \frac{1}{|\mathcal{C}_{\text{working}}(I)|}\sum_{C_j\in\mathcal{C}_{\text{working}}(I)}\psi_j(\hat{\mathcal{A}}_{S_j}),$$

which parallels the RL reward design used for propagation and restart scheduling in Section 6.

Constraint Projection and Feasibility Refinement. Predictions are refined by a differentiable projection operator:

$$\hat{\mathcal{A}}^{(t+1)} = \hat{\mathcal{A}}^{(t)} - \eta \, \nabla_{\hat{\mathcal{A}}} L_{\mathcal{C}_{\text{working}}}(\hat{\mathcal{A}}^{(t)}), \;\; L_{\mathcal{C}_{\text{working}}}(\hat{\mathcal{A}}) = \sum_{C_j \in \mathcal{C}_{\text{working}}(I)} [1 - \psi_j(\hat{\mathcal{A}}_{S_j})]^2,$$

where $\eta > 0$ is a step size. This mirrors conflict-resolution updates in `LCG`/`CDCL`, ensuring projection-based satisfaction rather than symbolic clause learning.

Sequential Policy Formulation. Alternatively, direct solving can be cast as a sequential decision policy $\pi_\phi(a_t \,|\, s_t)$ generating partial assignments:

$$s_t = (\mathcal{A}_t, \mathcal{C}^t_{\text{working}}), \quad a_t \in \{\texttt{assign}, \texttt{backtrack}\}.$$

Rewards follow the reduction in unsatisfied constraints:

$$r_t = |\mathcal{C}^{t-1}_{\text{working}}| - |\mathcal{C}^t_{\text{working}}|,$$

and π_ϕ is optimized by the `PPO` objective (defined by (16) in Appendix B.1). This MDP formalism is identical to that used for propagation pruning in Algorithm 3.

5.4 Transfer and Meta-Learning across `CSP` Families

When `CSP` instances arise from similar distributions, learned solvers can transfer knowledge. Given source and target families $(\mathcal{F}_{\text{src}}, \mathcal{F}_{\text{tgt}})$, adaptation minimizes

$$\min_{\theta'} \mathbb{E}_{I \sim \mathcal{F}_{\text{tgt}}} L(f_{\theta'}(I), \, \mathcal{A}_I) \quad \text{s.t.} \quad \|\theta' - \theta\|_2^2 \le \epsilon,$$

with $\epsilon > 0$ bounding deviation from pretrained weights θ. Meta-learning further optimizes for fast adaptation:

$$\theta'_k = \theta - \alpha \nabla_\theta L_{\mathcal{F}_k}(\theta), \qquad \min_\theta \sum_k L_{\mathcal{F}_k}(\theta'_k),$$

where $\alpha > 0$ is the inner learning rate.

Cross-Domain Generalization. Constraint-aware, binarized networks [30] generalize across structured `CSP` types (scheduling, resource allocation, etc.) by training on mixed distributions using shared encoders and consistency losses. This is analogous to solver-generalization effects observed in `SAT` meta-learners of Section 6.

In summary, direct solution learning reinterprets `CSP` solving as differentiable constraint satisfaction:

$$I \xrightarrow{f_\theta} \hat{\mathcal{A}} \xrightarrow{\Pi_{\mathcal{C}_{\text{working}}}} \mathcal{A}^*,$$

where f_θ produces a candidate assignment and $\Pi_{\mathcal{C}_{\text{working}}}$ ensures feasibility.

6 Learning for Classical Search

Classical search-based `SAT` solvers address a canonical NP-complete problem that lies at the core of modern combinatorial optimization and logical inference frameworks. They form the computational backbone for reasoning over Boolean formulas and underpin numerous applications in verification, planning, and automated deduction.

Goal and intuition. The goal of this section is to clarify how learning components can be incorporated into the different phases of traditional `SAT` solving, transforming fixed heuristic procedures into adaptive reasoning systems. The intuition is that each major operation of a solver—propagation, decision, conflict analysis, and restart—can be viewed as a decision process that can be optimized through data or experience. By learning to guide these phases, solvers can recognize recurring structural patterns across instances and dynamically adjust their strategies.

Motivation. Integrating ML and RL into classical `SAT` search provides a foundation for self-improving symbolic systems that retain logical soundness while benefiting from statistical generalization. This view reframes `SAT` solving as a sequence of learnable reasoning decisions, establishing the conceptual basis for the hybrid architectures described in later sections.

6.1 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 serves as both a fundamental model for logical reasoning and a computational bridge between symbolic and numerical optimization.

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

6.2 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. It is applied in Algorithm 3 (S1$_3$–S3$_3$). Given a satisfiable formula ϕ, the set of backbone variables is

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

Backbone variables form the structural core of ϕ and have a direct impact on solver efficiency: correctly predicting their fixed polarity prior to search reduces branching depth and the number of conflicts encountered during propagation.

Backbone Prediction via Monte Carlo Sampling. Following [59], a logistic regression model (defined by (15) in Appendix B.1) 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 ϕ_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 ϕ_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 ϕ_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 (see Algorithm 3).

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 (Algorithms 3–4) further refine these initial polarity estimates through online interaction with the solver.

6.3 Classical Algorithms and Related Solvers

This section reviews LCG and CDCL solver structures, including initialization, propagation, branching, and conflict analysis. It provides the base for Algorithm 1 (S0_1–S3_1), Algorithm 2 (S0_2–S3_2), and their ML-enhanced variants: Algorithm 3, 4, and 5.

State-of-the-art SAT solvers are largely based on two algorithmic paradigms:

- **Lazy Clause Generation (LCG)** – a hybrid method integrating CSP-style propagation with SAT-style clause learning;
- **Conflict-Driven Clause Learning (CDCL)** – a purely Boolean approach operating on CNF representations.

A detailed discussion of representative software implementing LCG and CDCL algorithms is presented in Appendix A.2.

We next present generic frameworks for both approaches. `LCG` (=Algorithm 1) illustrates the `LCG` procedure, which iteratively applies constraint propagation, heuristic decisions, and conflict analysis until either a satisfying assignment is found or infeasibility is proven.

(S0$_1$) Initialization. The solver initializes the partial assignment $\mathcal{A} := \emptyset$, sets $d := 0$, and defines the working set of constraints $\mathcal{C}_{\text{working}} := \mathcal{C}$. It also maintains a record `trial` of pairs (l_i, d) for each assigned literal l_i (either x_i or $\neg x_i$), and initializes a Boolean conflict indicator `conflict` $:= 0$.

(S1$_1$) Constraint propagation. For each constraint $c_k \in \mathcal{C}$, domains are reduced as

$$D'(x_i) = \{v \in \mathcal{D}(x_i) \mid c_k(x_i = v) \text{ holds under } \mathcal{A}\}.$$

If $D'(x_i) = \emptyset$ for some x_i, then a conflict is detected (`conflict` $:= 1$). If $D'(x_i) = \{v\}$, then the assignment $x_i = v$ is forced, and the assignment set is updated by

$$\mathcal{A} := \mathcal{A} \cup \{l_i = \text{true}\}, \qquad \texttt{trial} := \texttt{trial} \cup \{(l_i, d)\}.$$

Propagation continues until no further domain reductions are possible or a conflict arises.

(S2$_1$) Heuristic decision. If no conflict was detected in (S1$_1$), the solver selects an unassigned variable x_i and a value $v \in D'(x_i)$ according to a branching heuristic. The assignment

$$l_i = (x_i = v)$$

is added to the current assignment, and the decision level is increased:

$$\mathcal{A} := \mathcal{A} \cup \{l_i\}, \qquad d := d + 1, \qquad \texttt{trial} := \texttt{trial} \cup \{(l_i, d)\}.$$

(S3$_1$) Conflict analysis and backjumping. If a conflict is detected in (S1$_1$), the solver derives a learned clause c_{learned} from the conflicting constraints and augments the working set

$$\mathcal{C}_{\text{working}} := \mathcal{C}_{\text{working}} \cup \{c_{\text{learned}}\}.$$

It then computes the backjump level by examining the decision levels of the literals appearing in the learned clause c_{learned}. For simplicity, we define

$$d_{\text{backjump}} := \max\{d_i \mid l_i \in c_{\text{learned}}\},$$

that is, the highest decision level among its literals. In practice, modern `CDCL` solvers exclude the unique implication point (UIP) literal and use the *second-highest* decision level to ensure correct non-chronological backtracking. All assignments made at levels higher than d_{backjump} are then removed:

$$\mathcal{A} := \mathcal{A} \setminus \{l_i \mid d_i > d_{\text{backjump}}\}, \qquad d := d_{\text{backjump}}.$$

If $d_{\text{backjump}} < 0$, the solver declares the instance UNSAT; otherwise, propagation resumes from the updated level.

The loop (S1[1])–(S3[1]) continues until $\mathcal{C}_{\text{working}} = \emptyset$, in which case a satisfying assignment has been found and the solver returns SAT. If instead a conflict is detected at the root level (i.e., $d_{\text{backjump}} < 0$) and no further backtracking is possible, the solver declares UNSAT.

Termination states. The solver therefore terminates in one of two possible states:

- SAT — a satisfying assignment $\mathcal{A}$ has been found such that all constraints in $\mathcal{C}_{\text{working}}$ are satisfied;
- UNSAT — the solver has proven that no assignment $\mathcal{A}$ can satisfy all constraints, i.e., the instance is unsatisfiable.

These outcomes correspond to the classical decision problem formulation of Boolean satisfiability.

Algorithm 1 A Generic LCG Framework

```
Initialization
(S0[1]) Initialize $\mathcal{A}$, decision level $d$, working set $\mathcal{C}_{\text{working}}$, record trial, and conflict.
repeat
    Constraint propagation
    (S1[1]) Reduce domains, assign forced values, and detect conflicts.
    Heuristic decision
    (S2[1]) If no conflict, select a variable, assign a value, increase decision level, and continue.
    Conflict analysis
    (S3[1]) If conflict, derive a learned clause, backjump, and resume. If $d_{\text{backjump}} < 0$, declare UNSAT.
until all clauses in $\mathcal{C}_{\text{working}}$ are satisfied or a conflict is detected at the root level
```

CDCL (=Algorithm 2) differs from LCG mainly in the propagation step, since CDCL works purely on Boolean formulas encoded in CNF, while LCG combines SAT solving with CSP propagation. The overall structure is similar: initialization (S0_2), unit propagation (S1_2), heuristic decision (S2_2), and conflict analysis with backjumping (S3_2).

(S0_2) Initialization. The solver starts with an empty assignment $\mathcal{A} := \emptyset$, decision level $d := 0$, and the working clause set $\mathcal{C}_{\text{working}} := \mathcal{C} = \{c_1, c_2, \dots, c_m\}$. A record `trial` stores assignments and their decision levels, (l_i, d), where l_i is a literal. The conflict flag is initialized as $\texttt{conflict} := 0$.

(S1_2) Unit propagation and conflict detection. For each clause $c_j = (l_1 \vee l_2 \vee \cdots \vee l_k)$, check:
- If exactly one literal is unassigned under $\mathcal{A}$ and all other $k-1$ literals are false, i.e.

$$\big|\{l_i \mid l_i \text{ unassigned under } \mathcal{A}\}\big| = 1, \quad \big|\{l_i \mid l_i = 0 \text{ under } \mathcal{A}\}\big| = k - 1,$$

then the unassigned literal must be set to true. The assignment set is updated by

$$\mathcal{A} := \mathcal{A} \cup \{l_i = \text{true}\}, \qquad \texttt{trial} := \texttt{trial} \cup \{(l_i, d)\}.$$

- If all literals in a clause are false under $\mathcal{A}$, i.e.

$$\big|\{l_i \mid l_i = 0 \text{ under } \mathcal{A}\}\big| = k,$$

then the clause is falsified, a conflict is detected, and $\texttt{conflict} := 1$.

This process continues until no more unit clauses remain.

(S2_2) Heuristic decision. If no conflict was found, the solver chooses an unassigned variable x_i using a heuristic (e.g., VSIDS or CHB discussed in Section 7.1), assigns it a Boolean value, and increments the decision level:

$$\mathcal{A} := \mathcal{A} \cup \{l_i\}, \qquad d := d + 1, \qquad \texttt{trial} := \texttt{trial} \cup \{(l_i, d)\}.$$

(S3_2) Conflict analysis and backjumping. If a conflict occurs, the solver identifies a conflict clause c_{conflict} and uses resolution to derive a learned clause c_{learned}, which is added to the working set:

$$\mathcal{C}_{\text{working}} := \mathcal{C}_{\text{working}} \cup \{c_{\text{learned}}\}.$$

The solver then computes the backjump level

$$d_{\text{backjump}} := \max\{d_i \mid l_i \in c_{\text{learned}}\},$$

and removes all assignments at levels higher than d_{backjump}:

$$\mathcal{A} := \mathcal{A} \setminus \{l_i \mid d_i > d_{\text{backjump}}\}, \qquad d := d_{\text{backjump}}.$$

If $d_{\text{backjump}} < 0$, the formula is declared **unsatisfiable**. Otherwise, the solver resumes propagation.

The loop (S1[2])–(S3[2]) continues until $\mathcal{C}_{\text{working}} = \emptyset$, in which case the formula is satisfiable.

Algorithm 2 A Generic CDCL Framework

```
Initialization
(S0_2) Initialize $\mathcal{A}$, decision level $d$, working set $\mathcal{C}_{\text{working}}$, record trial, and conflict.
repeat
    Unit propagation
    (S1_2) Perform unit clause propagation and detect conflicts.
    Heuristic decision
    (S2_2) If no conflict, select a variable, assign a value, increase decision level, and continue.
    Conflict analysis
    (S3_2) If conflict, learn a clause, backjump, and resume. If $d_{\text{backjump}} < 0$, declare UNSAT.
until $\mathcal{C}_{\text{working}} = \emptyset$        % return SAT.
```

Literal Block Distance (LBD). This paragraph defines the LBD metric for evaluating clause quality during conflict learning. It is employed in Algorithm 5 (S0[5], S1[5]) to guide clause retention.

The Literal Block Distance (LBD) [5] is a clause-quality metric used in modern CDCL solvers to evaluate the relevance of learned clauses during conflict analysis. Let $C = (l_1 \vee l_2 \vee \cdots \vee l_k)$ be a learned clause, and let level(l_i) denote the decision level at which literal l_i was assigned. The LBD of clause C is defined as

$$\texttt{LBD}(C) = \big|\{\text{level}(l_i) \mid l_i \in C\}\big|,$$

that is, the number of distinct decision levels among the literals of C. Clauses with lower LBD values are considered more general and therefore more useful, since they connect fewer decision levels and tend to prune the search space more effectively. Consequently, LBD serves as a key heuristic for clause activity, retention, and deletion in high-performance solvers such as Glucose and Kissat.

6.4 NNs as Enhancers for CSP Solvers

This paragraph summarizes ML and RL integrations at the propagation, decision, and conflict phases of LCG/CDCL solvers. It is directly underlined by

Algorithm 3 (S0_3–S3_3), Algorithm 4 (S0_4–S3_4), and Algorithm 5 (S0_5–S3_5).

Building on the definitions in Section 3, data–driven techniques have been successfully integrated into the LCG and CDCL frameworks for CSP/SAT solving, including logistic or neural classifiers for backbone prediction, solver-selection models for instance classification, and PPO-based branching and restart strategies. These methods replace hand-crafted heuristics with adaptive learned policies, improving scalability and robustness; representative ML and RL enhancements are summarized in Table 4.

Table 4: ML and RL methods discussed for CSP/SAT solvers. These approaches enhance the classical LCG and CDCL frameworks at different solver stages.

Solver Component	**ML/RL Enhancement**	**References**
Propagation	Backbone variable prediction via logistic regression and Monte Carlo sampling; RL–guided pruning in domain-specific encodings; ML-driven abstraction–refinement using ILP relaxations	[17,35,59]
Heuristic Decision	Instance classification for solver selection; automated parameter tuning via learned performance models; Learning Rate Branching (LRB) with multi-armed bandits; classifier-driven and transformer-based branching; polarity prediction through logistic regression	[10, 11, 18, 32,50,59]
Conflict Analysis & Backjumping	Clause-utility prediction using regression or NNs; RL-based restart scheduling and adaptive backjumping; domain-specific RL for timetabling; ML-guided cryptanalytic solvers (NeuroGIFT)	[32,35,54]
Beyond Core Loop and Analysis	ML study of community-structured SAT phase transitions; SAT encodings for interpretable ML models such as optimal decision trees	[8,39]

The propagation, heuristic decision, and conflict analysis phases of LCG (Algorithm 1)/CDCL (Algorithm 2) correspond to the steps where ML or RL modules can be embedded. Equations defining logistic regression, GNN updates, and MDPs are omitted here since they are defined in Section 3; references below link directly to those foundations.

Step (S1$_1$)/(S1$_2$) of LCG/CDCL – Propagation. Propagation uses ML-predicted backbone variables and RL–guided pruning policies (definitions of policy π_θ and reward shaping follow Section 3). Algorithm 3 summarizes the augmented propagation procedure.

The integration of learned heuristics into the propagation stage is summarized in Algorithm 3. This module extends (S1$_1$) and (S1$_2$) by embedding ML–driven backbone estimation and RL–based pruning policies directly into the consistency enforcement phase. In (S0$_3$), statistical features are extracted from the CNF to initialize the logistic model. In (S1$_3$), a Monte Carlo refinement estimates variable confidence and fixes high-probability backbone literals before search. In (S2$_3$), an RL policy learns to prune infeasible domains, improving propagation efficiency in LCG. Finally, (S3$_3$) integrates both ML predictions and RL guidance into a unified propagation routine, reducing conflict depth and enabling faster convergence [35, 59].

Algorithm 3 ML-Enhanced Propagation for (S1$_1$)/(S1$_2$) of LCG/CDCL

```
Initialization
(S0$_3$) Extract CNF features and train or load a logistic model for backbone prediction.
Backbone Estimation
(S1$_3$) Use Monte Carlo sampling to estimate variable confidence; fix literals with high backbone probability.
RL–Guided Pruning
(S2$_3$) Model propagation as a decision process; apply a learned policy to prune infeasible domains.
Integration
(S3$_3$) Combine ML and RL guidance to accelerate propagation and reduce conflicts.
```

Figure 3 illustrates how the logistic model predicts backbone literals while an RL policy prunes infeasible domains during constraint propagation.

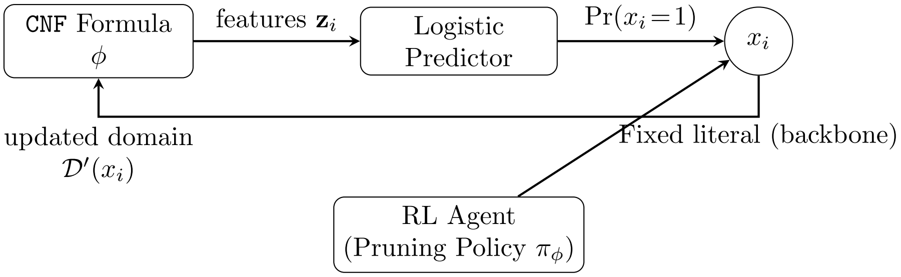


Figure 3: ML-enhanced propagation: the logistic model predicts backbone literals, while an RL agent prunes infeasible branches to accelerate propagation.

RL Formulation for Pruning. This paragraph formulates propagation pruning as an MDP with states $(\mathcal{A}_t, \mathcal{C}^t_{\text{working}})$, actions $\{\texttt{fix}, \texttt{prune}\}$, and reward r_t. It is implemented in Algorithm 3 (S2$_3$, S3$_3$).

The propagation process can be modeled as a Markov decision process (MDP)

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

where the state $s_t \in \mathcal{S}$ encodes the solver's internal configuration, including the current partial assignment and the working clause set, and the action $a_t \in \mathcal{A}$ corresponds to fixing or pruning a literal. Transitions follow the solver's propagation dynamics $\Pr(s_{t+1} \mid s_t, a_t)$, and the reward signal quantifies progress in constraint reduction, for example,

$$r_t = |\mathcal{C}^{t-1}_{\text{working}}| - |\mathcal{C}^t_{\text{working}}|,$$

which measures the decrease in the number of active clauses or conflicts.

Formally, the propagation phase defines

$$\begin{gathered} s_t = (\mathcal{A}_t,\, \mathcal{C}^t_{\text{working}}), \\ a_t \in \{\texttt{fix}(x_i = v),\ \texttt{prune}(x_i = v)\}, \\ \Pr(s_{t+1} \mid s_t, a_t) \text{ describes the solver's update dynamics}, \\ R_t = |\mathcal{C}^{t-1}_{\text{working}}| - |\mathcal{C}^t_{\text{working}}|, \\ \gamma \in (0,1) \text{ is the discount factor.} \end{gathered}$$

The policy $\pi_\phi(a_t \mid s_t)$, parameterized by ϕ, is trained using the PPO objective (defined by (16) in Appendix B.1) to maximize the expected discounted return $\mathbb{E}_{\pi_\phi}[\sum_t \gamma^t R_t]$, thus guiding the solver to prune infeasible branches efficiently.

In practical implementations, the state representation s_t is constructed from compact solver statistics—such as the number of active clauses, mean LBD, and current decision depth—or from neural embeddings of the working clause set $\mathcal{C}_{\text{working}}$ produced by a GNN encoder. The policy $\pi_\phi(a_t \mid s_t)$, parameterized by ϕ, is trained using the `PPO` objective (defined by (16) in Appendix B.1) to maximize the expected discounted return

$$\mathbb{E}_{\pi_\phi}\left[\sum_t \gamma^t r_t\right],$$

thereby learning an adaptive pruning strategy as formalized in Algorithm 3.

Step (S2$_1$)/ (S2$_2$) of `LCG`/`CDCL` – Heuristic Decision. Branching decisions use instance classification, parameter tuning, and policy optimization via `PPO` as described in Section 3. Equation-based definitions of classifiers and softmax predictors are omitted and replaced by references. Algorithm 4 summarizes the decision mechanism.

ML and RL enhance the heuristic decision phase by replacing static branching rules with adaptive, data–driven policies. Algorithm 4 extends (S2$_1$) and (S2$_2$) through instance classification, parameter tuning, and neural branching strategies. In (S0$_4$), `CNF` features are collected and classified to select solver-specific configurations. In (S1$_4$), parameters such as restart intervals and clause deletion thresholds are automatically tuned using learned performance models. In (S2$_4$), branching variables are chosen through multi-armed bandit learning or transformer-based scoring, integrating polarity prediction for conflict reduction. Finally, (S3$_4$) combines these models into an adaptive decision module that guides search direction and branching depth dynamically, achieving faster convergence and improved solver generalization [11, 18, 32, 50, 59].

Figure 4 shows adaptive branching, where ML classification and bandit-based or transformer-guided policies dynamically select variables and polarities.

Algorithm 4 ML-Enhanced Heuristic Decision for (S2$_1$)/(S2$_2$) of `LCG/CDCL`

Initialization

(S0$_4$) Extract `CNF` features and classify instance type to load solver configuration and parameter settings.

Parameter Optimization

(S1$_4$) Use learned performance models to adjust solver parameters (e.g., restart interval, clause deletion ratio) before search.

Learning-Based Branching

(S2$_4$) Select branching variables using learning-rate–based multi-armed bandits or Transformer-based scoring via the attention formulation (defined by (13) in Appendix B.1). Predict polarity using a classifier or logistic model (defined by (15) in Appendix B.1) to align initial assignments with backbone tendencies.

Integration and Result

(S3$_4$) Integrate classification, parameter tuning, and neural branching into the decision phase to guide variable selection dynamically while reducing conflicts. Achieves improved solver adaptability and faster convergence [11,18,32,50,59].

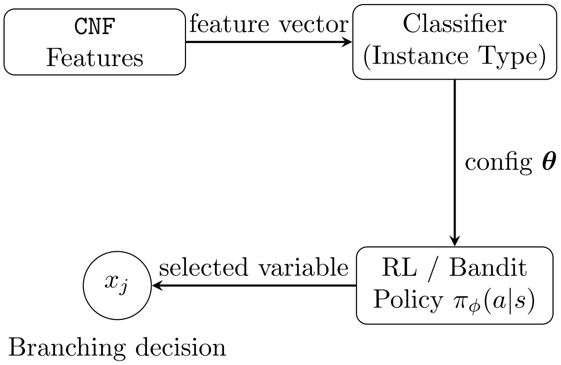

CNF Features
feature vector
Classifier (Instance Type)
config $\boldsymbol{\theta}$
RL / Bandit Policy $\pi_\phi(a|s)$
selected variable
x_j
Branching decision

Figure 4: Adaptive branching via ML classification and RL/bandit decision policies. The model selects the next variable and polarity based on learned solver statistics.

Step (S3$_1$)/ (S3$_2$) of `LCG/CDCL` – Conflict Analysis and Backjumping. Conflict resolution benefits from ML-based clause utility prediction and RL-guided restart scheduling (as in the `PPO` framework introduced earlier). Algorithm 5 details the procedure.

ML further refines the conflict analysis and backjumping phase by predicting clause utility, guiding restart decisions, and prioritizing search in specialized domains. Algorithm 5 extends (S3_1) and (S3_2) through clause-utility prediction, RL-driven restart scheduling, and neural-guided search in cryptanalytic problems. In (S0_5), clause statistics such as size, age, and LBD are collected after each conflict. In (S1_5), a regression or neural model predicts the usefulness of learned clauses, allowing low-utility clauses to be deprioritized or removed. In (S2_5), RL agents dynamically determine restart or continuation actions based on solver states, improving restart efficiency and clause quality. In (S3_5), domain-specific neural prioritization models, such as NeuroGIFT, adapt the solver's backtracking strategy for cryptanalysis, ranking candidate assignments by plausibility. This combination of predictive and adaptive control significantly reduces redundant conflicts and enhances solver robustness [32, 35, 54].

Algorithm 5 ML-Enhanced Conflict Analysis and Backjumping for (S3_1)/(S3_2) of LCG/CDCL

Initialization

(S0_5) Collect features of learned clauses (e.g., size, LBD, activity, age) after each conflict.

Clause Utility Prediction

(S1_5) Use a regression or neural model to estimate clause utility. Retain high-utility clauses and remove or deprioritize those predicted to be less useful.

RL-Guided Restart Scheduling

(S2_5) Represent the solver state as a decision process with actions {restart, continue}. Train an RL policy using the PPO objective (defined by (16) in Appendix B.1) to maximize runtime efficiency by selecting optimal restart moments based on conflict patterns.

Neural Prioritization and Integration

(S3_5) Integrate domain-specific models such as NeuroGIFT to prioritize search branches or candidate assignments using neural scoring. Combine predictions with a restart policy for adaptive backjumping and faster convergence.

Result

ML models improve clause retention and restart scheduling, while neural prioritization accelerates key recovery and structured search [32, 35, 54].

Figure 5 depicts conflict analysis enhanced by ML clause-utility prediction and RL-based restart scheduling to improve solver efficiency.

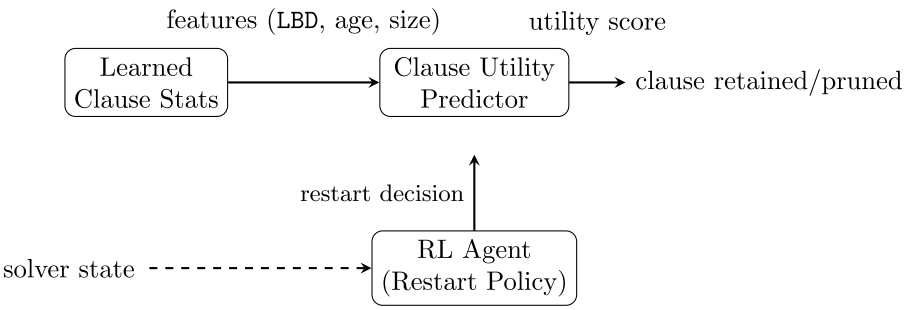


Figure 5: Conflict analysis enhanced by ML-based clause evaluation and RL-guided restart scheduling.

RL Formulation for Restart Scheduling. This paragraph defines restart scheduling as an MDP over solver statistics with actions {`restart`, `continue`} and reward based on conflict reduction. It is used in Algorithm 5 (S2$_5$, S3$_5$).

Restart scheduling can be modeled as a Markov decision process (MDP)

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

where the state $s_t \in \mathcal{S}$ encodes solver statistics such as the current number of conflicts, the mean `LBD` value $\overline{\text{LBD}}_t$, and the current decision level d_t. The available actions are $a_t \in \{\texttt{restart}, \texttt{continue}\}$, and transitions follow the solver's internal update dynamics $\Pr(s_{t+1} \mid s_t, a_t)$. The reward encourages reductions in the conflict rate:

$$r_t = \Delta_{\text{conflict}}^{t-1} - \Delta_{\text{conflict}}^{t},$$

where $\Delta_{\text{conflict}}^{t}$ denotes the average number of conflicts per decision window. A policy $\pi_\phi(a_t \mid s_t)$, parameterized by ϕ, is trained using the `PPO` objective (defined by (16) in Appendix B.1) to maximize the expected discounted return

$$\mathbb{E}_{\pi_\phi}\left[\sum_t \gamma^t r_t\right],$$

thereby providing an adaptive mechanism for determining optimal restart points based on real-time solver behavior.

Neural Prioritization Model (`NeuroGIFT`). This paragraph describes `NeuroGIFT`, a neural encoder–decoder framework that ranks candidate assignments for cryptanalytic `SAT` solving. It is implemented in Algorithm 5 (S3$_5$).

`NeuroGIFT` employs a neural encoder–decoder architecture to prioritize candidate assignments during cryptanalytic search. The encoder maps the variable–clause adjacency matrix into hidden embeddings, while the decoder produces a ranked list of promising variable assignments that are fed back into the solver as branching priorities. Input features represent Boolean variables and clause dependencies, which are processed through multilayer perceptrons to yield scores indicating assignment likelihood. These scores are iteratively refined using feedback from solver conflicts, integrating neural ranking with traditional CDCL backjumping. This hybrid design enables adaptive prioritization while preserving solver soundness.

Beyond the Core Loop. This paragraph discusses ML for analyzing SAT community structures and learning interpretable decision trees using SAT encodings. It is applied in Algorithm 6 ($S0_6$–$S3_6$).

ML has also been used to study the structure of SAT formulas themselves. Community-structured SAT formulas exhibit thresholds different from random SAT, and ML-based analysis has been instrumental in understanding satisfiability phase transitions in such structured cases [8]. Furthermore, SAT solving itself can serve ML: for instance, SAT encodings have been developed for learning *optimal decision trees*, yielding compact, interpretable models that contribute explainable AI [39].

Beyond the core solving loop, ML contributes to structural analysis and interpretable model construction. Algorithm 6 extends this perspective by incorporating ML-based structural study and SAT-encoded model learning. In ($S0_6$), the solver extracts graph-level and community-based statistics from SAT instances to characterize problem structure. In ($S1_6$), these representations are used to identify community organization and predict phase transition regions. In ($S2_6$), SAT-based encodings are employed to learn interpretable models such as optimal decision trees. Finally, ($S3_6$) integrates these insights to guide both solver analysis and the synthesis of explainable ML models [8, 39].

Algorithm 6 ML Applications Beyond the Core Loop of LCG/CDCL

Initialization

($S0_6$) Collect global CNF features and construct a clause–variable or community graph representation.

Structural Analysis

($S1_6$) Use ML models to detect community structures and estimate satisfiability thresholds in complex SAT instances.

SAT-Encoded Learning

($S2_6$) Encode interpretable models such as optimal decision trees as CNF formulas and solve them using SAT to obtain minimal, consistent models.

Integration and Result

($S3_6$) Combine structural insight with SAT-encoded model learning to enhance solver analysis and contribute explainable, hybrid neuro-symbolic reasoning frameworks [8, 39].

Figure 6 visualizes meta-level learning, where ML models analyze SAT structure and construct interpretable decision-tree models via SAT encodings.

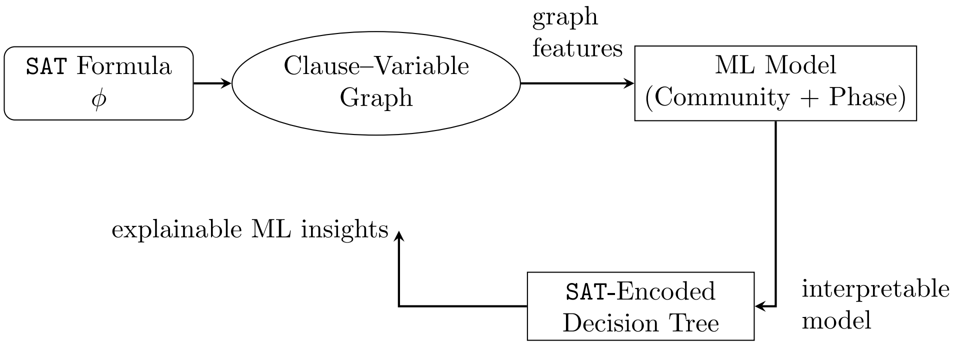


Figure 6: Beyond the solving loop: ML models extract community structure and encode interpretable decision trees via SAT formulations.

Community Structure Metrics. This paragraph defines modularity Q as a measure of community structure in SAT clause–variable graphs. It is employed in Algorithm 6 (S1$_6$).

Given a CNF formula ϕ, let $\mathcal{G} = (V_{\text{var}} \cup V_{\text{cl}}, E)$ denote its bipartite clause–variable graph, where V_{var} and V_{cl} represent variable and clause nodes, respectively. The community structure of $\mathcal{G}$ is quantified by the modularity measure

$$Q = \frac{1}{2|E|} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2|E|} \right] \delta(c_i, c_j),$$

where A_{ij} is the adjacency matrix, k_i is the degree of node i, c_i its community label, and $\delta(\cdot,\cdot)$ is the Kronecker delta. For simplicity, the clause–variable bipartite graph is treated as undirected when computing the modularity Q. For bipartite graphs $\mathcal{G}$, node degrees are normalized such that $\sum_i k_i = 2|E|$, ensuring that the modularity Q remains properly scaled under the clause–variable partition. ML models leverage Q and related graph-structural features to predict satisfiability thresholds and phase-transition behavior in structured SAT instances.

SAT-Encoded Optimal Decision Trees. This paragraph presents SAT-encoded optimal decision-tree learning, ensuring logical consistency between sample labels and tree paths. It is used in Algorithm 6 (S2$_6$, S3$_6$).

Learning an optimal decision tree of depth d can be formulated as a SAT problem. Each internal node corresponds to a Boolean variable z_j representing a split predicate, and each leaf encodes a class label y_ℓ. For a dataset $\{(x_i, y_i)\}_{i=1}^N$, the CNF encoding enforces label consistency between samples and tree structure as

$$\bigwedge_{i=1}^{N} \bigvee_{\ell \in L} \Big((x_i \models \mathrm{path}(\ell)) \to (y_i = y_\ell) \Big),$$

where each path(ℓ) represents a conjunction of split predicates leading to leaf ℓ with class label y_ℓ. Equivalently, the constraint can be expressed as

$$y_i = f_{\mathrm{tree}}(x_i; z_1, \ldots, z_d), \quad \forall i \in [N],$$

where f_{tree} is the Boolean function realized by the candidate tree. Solving the resulting CNF yields a tree of minimal size that satisfies all label constraints, thereby producing an interpretable, globally optimal classifier learned through logical inference.

Table 5 summarizes how ML and RL enhance the main stages of both Algorithm 1 (LCG) and Algorithm 2 (CDCL). Each solver phase benefits from newly integrated heuristics, predictive models, or reinforcement strategies that strengthen propagation, decision making, and conflict analysis. Beyond the core solving loop, ML further contributes to structural analysis of SAT formulas and to SAT-based encodings for interpretable model learning. The table also complements Table 4 by detailing, for each major solver step in LCG and CDCL, the specific ML and RL methods that drive performance improvements and solver adaptability.

Neural Architectures for SAT Solving. Neural architectures for SAT solving extend the graph-based and attention-based foundations established in Section 3. Given a clause–literal bipartite graph $\mathcal{G} = (V_{\mathrm{lit}} \cup V_{\mathrm{cl}}, E)$ representing a Boolean formula in CNF, each literal $v \in V_{\mathrm{lit}}$ and clause $c \in V_{\mathrm{cl}}$ is assigned an embedding $h_v, h_c \in \mathbb{R}^d$ that is iteratively refined through message passing (defined by Eq. (11) in Appendix B.1). The following neural models specialize this formulation:

- NeuroSAT [49] formulates satisfiability prediction as a binary classification problem over the entire clause–literal graph. At each iteration k, literal and clause embeddings are updated by

  $$h_c^{(k+1)} = \mathrm{MLP}_c\left(\sum_{v \in N(c)} h_v^{(k)}\right), \qquad h_v^{(k+1)} = \mathrm{MLP}_v\left(h_v^{(k)}, \sum_{c \in N(v)} h_c^{(k+1)}\right),$$

  followed by a graph-level pooling $\hat{y} = \sigma\big(\mathbf{w}^\top \sum_v h_v^{(K)}\big)$ that predicts whether the formula is satisfiable.

Table 5: ML/RL Enhancements in LCG and CDCL Solver Steps

Step	**ML/RL Techniques**	**References**
$(S1_1)$ / $(S1_2)$: Propagation	• Backbone prediction via logistic regression & Monte Carlo; • RL–guided pruning in domain-specific encodings; • Hybrid abstraction–refinement using ILP relaxations guided by ML	[59] [35] [17]
$(S2_1)$ / $(S2_2)$: Heuristic Decision	• Instance classification via CNF features; • Automated parameter tuning with ML; • Learning–based branching (LRB, SGD, etc.); • Classifier-driven branching; • Transformer-based variable selection; • Polarity prediction (default truth values)	[18] [11] [32] [10] [50] [59]
$(S3_1)$ / $(S3_2)$: Conflict Analysis & Backjumping	• ML-guided clause utility prediction; • RL-inspired restart policies; • domain-specific RL guidance (timetabling); ML-enhanced SAT solvers in cryptanalysis (NeuroGIFT)	[32] [35] [54]
Beyond the Core Loop	ML analysis of community-structured SAT thresholds; SAT encodings for learning optimal decision trees (XAI)	[8] [39]

- NeuralSAT [48] extends this idea by coupling unsatisfiable-core prediction with clause-level attention. For each clause embedding $h_c^{(K)}$, an attention score $\alpha_c = \mathrm{softmax}(\mathbf{q}^\top h_c^{(K)})$ estimates its likelihood of belonging to an unsat core. The network minimizes a cross-entropy loss $L_{\mathrm{unsat}} = -\sum_c [y_c \log \alpha_c + (1 - y_c) \log(1 - \alpha_c)]$, guiding solver heuristics such as clause deletion and restart policies.

- SATFormer [51] replaces local message passing with global multi-head attention (defined by Eqs. (13)–(14) in Appendix B.1) applied to tokenized clause–literal embeddings. The model computes $\mathrm{Attn}(Q, K, V) = \mathrm{softmax}(QK^\top/\sqrt{d_k})V$ over all variable–clause pairs, enabling long-range relational reasoning. A transformer decoder then produces satisfiability logits or variable activity scores used for branching and clause ranking.

The learning-enhanced SAT methods summarized in Table 5 focus primarily on classical ML and RL heuristics embedded within LCG/CDCL frameworks. NeuroSAT, NeuralSAT, and SATFormer generalize these ideas to end-to-end differentiable architectures that operate directly on clause–literal graphs using message passing and self-attention. These models form the conceptual bridge between the solver-embedded heuristics reviewed above and the neural–symbolic reasoning frameworks introduced in the next section on first-order and relational CSPs.

7 Learning for Heuristic Search

Heuristic search methods provide scalable strategies for navigating the immense combinatorial spaces encountered in constraint and satisfiability problems. They operate by assigning scores or priorities to candidate variable assignments, guiding the solver toward regions of the search space that are more likely to yield feasible or optimal solutions. In CSP/SAT solvers, these heuristics determine branching order, polarity, and restart timing—factors that critically affect runtime efficiency and convergence behavior.

Goal and intuition. The goal of this section is to illustrate how classical heuristic mechanisms can be augmented or replaced by learning-based models that adapt dynamically to problem structure and solver feedback. The intuition is that heuristic control—long regarded as a fixed, rule-driven component—can itself become a learnable function that captures statistical regularities in search dynamics. By interpreting branching and restart strategies as policy-learning problems, solvers can continuously refine their exploration behavior from experience.

Motivation. Learning-driven heuristics represent a key step toward adaptive reasoning systems that self-tune their decision strategies while preserving the completeness of symbolic search. They bridge the gap between handcrafted scoring rules and data-informed policies, forming the core of the hybrid architecture presented in Algorithm 4 ($S2_4$), where reinforcement and supervised learning jointly guide variable and clause selection.

This subsection reviews the operational principles of heuristic search and its extensions through learning-based frameworks. Classical scoring schemes such as `VSIDS` and `CHB` define rule-based priorities for variable selection, while their modern counterparts employ data-driven or reinforcement learning models that adaptively infer effective branching and restart strategies as described in Algorithm 4 ($S2_4$).

7.1 Policy-Based Heuristic Generation

Classical solvers employ fixed heuristics such as Variable State Independent Decaying Sum (`VSIDS`) [37] and Conflict History-Based (`CHB`) [33]. Learning–based approaches reinterpret these scoring functions as adaptive policies $\pi_\phi(a_t \,|\, s_t)$ over solver states $s_t = (\mathcal{A}_t, \mathcal{C}^t_{\text{working}})$.

The `VSIDS` heuristic dynamically prioritizes variables by assigning and updating activity scores based on their occurrence frequency in recently learned clauses, as detailed in Algorithm 7.

The `CHB` heuristic adaptively 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 Algorithm 8.

Both heuristics embody implicit reinforcement signals—conflict participation or resolution—suggesting a natural extension toward explicit policy learning.

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.

Algorithm 7 Variable State Independent Decaying Sum (`VSIDS`) Heuristic

Initialization

(S0$_7$) Initialize all variable scores to zero: $\text{score}(x_i) \leftarrow 0$ for all $x_i \in \mathcal{X}$.

Score Update

(S1$_7$) Each time a new clause C_{learned} is learned, increment the score of each variable appearing in the clause: $\text{score}(x_i) \leftarrow \text{score}(x_i) + 1$ for all $x_i \in C_{\text{learned}}$.

Decay Step

(S2$_7$) Periodically decay all scores using a multiplicative decay factor $\beta \in (0, 1)$ (typically $\beta = 0.95$): $\text{score}(x_i) \leftarrow \beta \cdot \text{score}(x_i)$ for all $x_i \in \mathcal{X}$.

Variable Selection

(S3$_7$) Select the next branching variable as the unassigned variable with the maximum score:

$$x^* = \underset{x_i \in \mathcal{X}_{\text{unassigned}}}{\text{argmax}} \ \text{score}(x_i).$$

Assignment Phase

(S4$_7$) Assign x^* using polarity caching, reusing its last successful truth value. If no prior polarity is available, assign randomly or using a default heuristic.

Iteration

(S5$_7$) Repeat steps (S1$_7$)–(S4$_7$) throughout the solver loop, allowing the scores to evolve dynamically as new clauses are learned and conflicts are resolved.

Algorithm 8 Conflict History Based (CHB) Heuristic

Initialization

(S0$_8$) Initialize an effectiveness counter for each variable: $\text{eff}(x_i) \leftarrow 0$ for all $x_i \in \mathcal{X}$.

Conflict Tracking

(S1$_8$) During search, record whether assigning a variable x_i contributes to resolving a conflict. Maintain a per-variable history of effectiveness events.

Reward Update

(S2$_8$) Each time the assignment of x_i helps resolve a conflict, increase its effectiveness counter: $\text{eff}(x_i) \leftarrow \text{eff}(x_i) + 1$.

Variable Selection

(S3$_8$) Select the next branching variable as the unassigned variable with the highest effectiveness score:

$$x^* = \underset{x_i \in \mathcal{X}_{\text{unassigned}}}{\text{argmax}} \ \text{eff}(x_i).$$

Assignment Phase

(S4$_8$) Assign the chosen variable x^* using polarity caching as in the VSIDS heuristic (Algorithm 7), reusing its last successful truth value.

Iteration

(S5$_8$) Repeat the process as conflicts are detected and resolved, allowing the effectiveness counters to evolve adaptively during search.

7.2 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 $\Pr(s_{t+1} \mid s_t, a_t)$ 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 (Algorithms 3, 5).

Policy Optimization. The policy parameters ϕ are optimized via the `PPO` objective (defined by (16) in Appendix B.1),

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

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 β 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

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

(consistent with Eq. (11), defined in Appendix B.1). Aggregated embeddings $h_v^{(K)}$ form the feature input to π_ϕ, yielding adaptive scores analogous to `VSIDS`/`CHB` but learned end-to-end.

Integration with Solvers. In practice, learned policies replace or modulate the scoring functions in step (S2$_1$)/(S2$_2$) of Algorithms 1–2. 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.

7.3 Evaluation Metrics and Benchmark Results

Learning–based heuristics are typically evaluated on standardized `SAT/CSP` benchmarks such as `SATCOMP`, `MiniZinc Challenge`, or synthetic instance families. Performance is compared against classical `VSIDS` and `CHB` baselines using the following metrics:

- **Runtime:** Average CPU time to reach `SAT/UNSAT` conclusions.
- **Conflicts:** Total number of conflicts encountered before termination.
- **Decision Depth:** Average decision level at solution or conflict.
- **Clause Quality:** Mean `LBD` value of learned clauses.
- **Generalization:** Performance 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 [33, 50, 59]. Neural policies consistently outperform fixed-parameter heuristics in dynamic or non-stationary problem families, while maintaining solver correctness.

Heuristic search thus transitions from static rule-based scoring to adaptive, policy-driven control. By embedding learning objectives directly into variable selection and propagation, modern solvers achieve amortized performance gains and cross-domain generalization,bridging classical `VSIDS/CHB` schemes with reinforcement-optimized decision frameworks.

8 GNNs for Graph Optimization

Graph optimization provides a natural interface between combinatorial optimization and modern learning-based reasoning. Many canonical `CSP` and `SAT` formulations—such as shortest path, spanning tree, max flow, and graph coloring—can be represented as optimization problems on graphs. This section first revisits these classical graph formulations to illustrate the geometric and algebraic structure underlying combinatorial reasoning, and then demonstrates how GNNs extend these ideas into a differentiable, learning-based framework.

Goal and intuition. The goal of this section is to bridge traditional graph-theoretic optimization and GNN-based learning architectures. The intuition is that GNNs replicate the logic of classical graph algorithms—propagation, aggregation, and constraint satisfaction—through learnable message-passing mechanisms that capture the same relational dependencies among nodes and edges.

This correspondence enables neural models to approximate or generalize well-established combinatorial solvers while operating directly on graph data.

Motivation. By integrating GNNs into the study of graph optimization, we highlight how structured neural computation provides an effective substrate for hybrid symbolic–statistical reasoning. GNNs thus act as a unifying model class that embeds optimization constraints within learned representations, enabling scalable and adaptive solutions to complex network-based decision problems.

The formal definitions and mathematical formulations of the classical graph optimization problems— such as the shortest path, minimum spanning tree, maximum flow, minimum cut, graph coloring, and traveling salesman—are provided in Appendix C.2, where each is expressed as a linear or mixed-integer program forming the foundation of the GNN learning framework developed here. Figure 7 summarizes these core problems and their relationships, illustrating how diverse combinatorial tasks can be unified under a common graph–optimization structure widely studied in operations research [29].

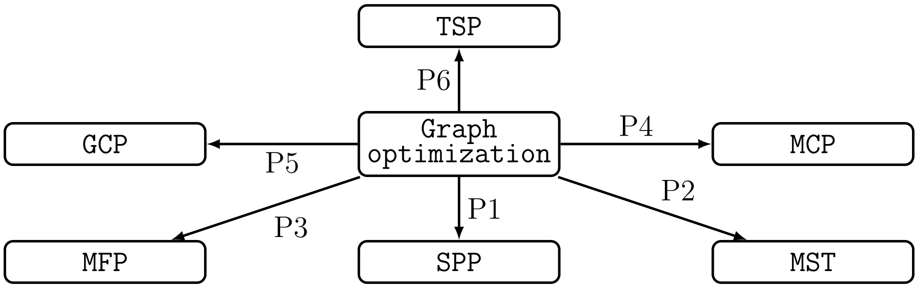


Figure 7: Flowchart for core graph optimization problems (P1–P6). SPP–shortest path, MST–minimum spanning tree, MFP– maximum flow, MCP–minimum cut, GCP–graph coloring, and TSP–traveling salesman. These canonical problems serve as the analytical foundation for GNN-based optimization models.

8.1 Overview of Graph Optimization Problems

Beyond integer formulations, another common representation for combinatorial optimization problems is the **graph**. A graph G is defined as a pair (V, E), where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices (or nodes) and $E = \{(v_i, v_j) \mid v_i, v_j \in V, v_i \neq v_j\}$ is the set of edges. Each edge $(v_i, v_j) \in E$ may indicate the presence of a connection ($e_{ij} \in \{0, 1\}$), a numerical weight ($e_{ij} \in \mathbb{R}$), or an attribute such as a color ($e_{ij} \in \mathbf{c} = \{\texttt{blue}, \texttt{red}\}$). See [14] for a comprehensive monograph on graph theory.

Graphs can be undirected or directed. In an undirected graph, $e_{ij} = 1$ implies $e_{ji} = 1$. Undirected graphs typically represent symmetric relationships, while directed graphs (or "digraphs") represent asymmetric interactions—such as influence in a statistical model or the flow of a commodity in a network. Cyclicity is an important concern in digraphs, and a Directed Acyclic Graph (DAG), wherein there are no cycles, is often sought to define the structure of influence in a model, wherein a cycle can indicate the model being vacuous. See [7] for a comprehensive text on digraphs. A recent deep line of work explores the cohomology of digraphs; see [23, 31].

Combinatorial optimization problems on graphs can typically be redefined in terms of some mixed-integer problem, indeed with e_{ij} treated as a binary or real-valued decision variable. However, while such a translation can enable the use of generic integer programming software to solve such problems, one should be careful before resorting to this "brute force" type of approach. The structure of graphs yields insight into considerably faster algorithms that use graph properties, which include a host of higher-order geometry, e.g., cliques, that can be informative in the design and structure of algorithms. See the text [29] for further details on combinatorial optimization.

8.2 GNN Architectures for Graph Problems

GNNs provide a unified framework to model and solve combinatorial optimization problems on graphs. Given a graph $G = (V, E)$ with node features $\{h_v^{(0)}\}_{v \in V}$ and edge features $\{e_{uv}\}_{(u,v)\in E}$, the GNN performs iterative *message passing* to update node and edge embeddings.

A general GNN layer follows the form:

$$h_v^{(k+1)} = \phi_{\text{upd}}\Big(h_v^{(k)}, \sum_{u \in \mathcal{N}(v)} \phi_{\text{msg}}(h_u^{(k)}, e_{uv})\Big), \tag{6}$$

where $\mathcal{N}(v)$ denotes the neighborhood of node v, and $\phi_{\text{msg}}, \phi_{\text{upd}}$ are learnable functions implemented by multilayer perceptrons (MLPs). After K iterations, the final embeddings $h_v^{(K)}$ capture multi-hop structural dependencies such as path lengths, connectivity, or capacities.

The learned embeddings are used by a decoder g_θ to produce task-specific outputs:

$$\hat{y}_v = g_\theta(h_v^{(K)}), \quad \text{or} \quad \hat{y}_{uv} = g_\theta([h_u^{(K)}; h_v^{(K)}; e_{uv}]),$$

depending on whether the problem requires node-level or edge-level predictions. Here, $[\,\cdot\,;\,\cdot\,]$ denotes vector concatenation.

Variants. Different GNN architectures specialize in distinct graph properties:

- **Graph Convolutional Networks (GCN)** [28]: use degree-normalized linear aggregation of neighboring node features.
- **Graph Attention Networks (GAT)** [57]: introduce learnable attention coefficients α_{uv} to weight incoming messages adaptively.
- **Graph Isomorphism Networks (GIN)** [60]: maximize representational power by learning injective aggregation functions.
- **Message-Passing NNs (MPNN)** [22]: generalize the above by conditioning messages on edge attributes e_{uv} and allowing continuous feature updates.

Training Objectives. GNNs for combinatorial optimization are trained under either:

1. **Supervised learning:** minimizing $\ell(\hat{y}, y^*)$ against ground-truth solutions (e.g., optimal paths or trees).
2. **Reinforcement learning:** maximizing the expected reward $\mathbb{E}[R]$ through environment interactions, where R reflects the quality (e.g., inverse cost or feasibility) of the constructed solution.

The differentiable structure of Eq. (6) allows gradient-based learning of global combinatorial relationships without explicit enumeration.

Complexity and Expressivity. Theoretical results show that GNNs can emulate classical algorithms such as Bellman–Ford (for shortest paths) or Prim's algorithm (for spanning trees) by embedding graph operations into differentiable message functions. Attention mechanisms enhance scalability by learning sparse dependencies across large graphs and selectively propagating information across critical edges.

8.3 Applications

The practical strength of GNNs emerges in their ability to approximate classical graph algorithms across a range of optimization tasks.

GNN-based solvers have been proposed for various graph optimization tasks, often trained on synthetic or real network instances where optimal or near-optimal solutions are known. The following subsections describe how GNN architectures approximate the core graph problems introduced earlier.

8.3.1 GNNs for SPP

In SPP, a GNN predicts edge-selection probabilities representing the likelihood of each edge belonging to the optimal path.

Let (s,t) denote the source and target nodes. Node embeddings $h_v^{(K)}$ are processed by a decoder that estimates distance potentials:

$$\hat{d}_v = g_\theta(h_v^{(K)}), \quad \forall v \in V.$$

The predicted path $\hat{P}_{s,t}$ is obtained by greedy edge selection following decreasing potential differences:

$$\hat{P}_{s,t} = \{(u,v) \in E : \hat{d}_u - \hat{d}_v \approx c_{uv}\},$$

where c_{uv} is the edge cost. Training minimizes the mean absolute error between predicted and true shortest distances d_v^* (computed by Dijkstra's algorithm):

$$L_{\mathrm{SPP}} = \frac{1}{|V|} \sum_{v \in V} |\hat{d}_v - d_v^*|.$$

GNNs trained in this manner achieve near-optimal routing on unseen graphs and generalize across variable graph sizes.

8.3.2 GNNs for MST Learning

For the MST, each edge (i,j) is assigned a probability $\hat{p}_{ij}$ of inclusion in the tree:

$$\hat{p}_{ij} = \sigma\big(g_\theta([h_i^{(K)}; h_j^{(K)}; e_{ij}])\big),$$

where $\sigma(\cdot)$ denotes the sigmoid activation. The loss encourages the selection of low-cost edges that maintain connectivity while avoiding cycles:

$$L_{\mathrm{MST}} = \sum_{(i,j) \in E} c_{ij}\hat{p}_{ij} + \lambda_1\, \mathrm{CyclePenalty}(\hat{p}) + \lambda_2\, \mathrm{DisconnectPenalty}(\hat{p}),$$

where $\lambda_1, \lambda_2 > 0$ are penalty weights. Here, $\mathrm{CyclePenalty}(\hat{p})$ penalizes edge selections that create cycles, and $\mathrm{DisconnectPenalty}(\hat{p})$ penalizes disconnected components. This formulation mimics Kruskal's algorithm while providing a differentiable, end-to-end relaxable variant. Empirically, trained GNNs recover near-optimal trees and scale linearly with graph size.

8.3.3 GNNs for MFP/MCP Estimation

For MFP, edge embeddings encode both capacity u_{ij} and direction. A message-passing network predicts flow magnitudes $\hat{f}_{ij}$ satisfying approximate conservation:

$$\hat{f}_{ij} = \mathrm{ReLU}\big(g_\theta([h_i^{(K)}; h_j^{(K)}; u_{ij}])\big), \quad \sum_j (\hat{f}_{ij} - \hat{f}_{ji}) \approx 0, \ \forall i \in V \setminus \{s, t\}.$$

The training objective minimizes violation of capacity and conservation while maximizing total feasible flow:

$$L_{\text{flow}} = \sum_{(i,j)\in E} \big[\max(0, \hat{f}_{ij} - u_{ij})\big]^2 + \sum_{i\neq s,t} \Big(\sum_j (\hat{f}_{ij} - \hat{f}_{ji})\Big)^2 - \eta \sum_j \hat{f}_{sj},$$

where $\eta > 0$ is a reward coefficient promoting larger feasible flows. MCP estimates can be derived from thresholded flow embeddings, consistent with the MFP/MCP duality.

8.3.4 GNNs for TSP Approximation

In the TSP, the model sequentially constructs a tour using an attention-based GNN (or Transformer) that selects the next node conditioned on the current partial route. Given node embeddings $\{h_v^{(K)}\}$, the policy network outputs selection probabilities:

$$\pi_\phi(v_t \mid s_t) = \mathrm{softmax}\big(h_{v_t}^{(K)\top} W_\phi \bar{h}_t\big),$$

where π_ϕ is the policy parameterized by ϕ, W_ϕ is a learned projection matrix, and $\bar{h}_t$ is the context vector summarizing the already visited nodes. The expected tour cost is minimized via

$$L_{\text{TSP}} = \mathbb{E}_{\pi_\phi}\Big[\sum_{(i,j)\in\text{tour}} c_{ij}\Big],$$

where c_{ij} denotes the travel cost between cities i and j. Training employs reinforcement learning with reward $R = -$tour length, optimized via policy-gradient or PPO updates. This yields near-optimal tours competitive with classical heuristics (e.g., Lin–Kernighan) on moderate-size instances.

GNNs thus act as differentiable approximators of classical graph algorithms. Their message-passing architecture (Eq. (6)) captures structural information directly from graph topology, enabling amortized inference on unseen instances. When coupled with reinforcement objectives or hybrid CSP solvers, they provide a powerful, learning–based approach for large-scale combinatorial graph optimization.

9 Learning for DisP

DisP provides a unifying framework for optimization problems that involve explicit logical alternatives among constraints. Originally introduced by Balas [6], it formalizes the structure of decisions that follow "either–or" or "one-of-many" logic, creating a bridge between symbolic reasoning and continuous optimization. This section revisits the classical foundations of DisP to prepare for its integration with machine learning and reinforcement learning methods capable of approximating or reasoning over disjunctive structures.

Goal and intuition. The goal of this section is to clarify how logical disjunctions can be expressed as optimization objects and how their convexifications enable tractable computation. The intuition is that each disjunction defines a set of feasible regions—each locally convex but globally nonconvex—whose union captures the combinatorial complexity of many real-world decision problems. By understanding this representation, we can later introduce learning mechanisms that approximate or guide the resolution of these logical alternatives.

Motivation. DisP serves as the theoretical foundation for modern mixed-integer and hybrid optimization, making it a natural candidate for neural or learning-based extensions. Learning-augmented DisP models enable systems to discover useful cuts, branching rules, or feasibility patterns automatically, combining the logical expressiveness of disjunctive reasoning with the adaptability of data-driven methods.

9.1 Technical Formulation of DisP

DisP, originally introduced by Balas [6] 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 [6].

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

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & x \in C_{\text{DisP}}, \end{array} \tag{7}$$

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

$$C_{\text{DisP}} := \left\{ x \in \mathcal{X} \mid \bigvee_{l=1}^{L} \big(g^l(x) = 0,\ h^l(x) \leq 0\big),\ x_i \in s_i\mathbb{Z} \text{ for } i \in I \right\}, \tag{8}$$

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 (8) can be expressed abstractly as

$$(g^1(x) = 0,\ h^1(x) \leq 0) \vee (g^2(x) = 0,\ h^2(x) \leq 0) \vee \cdots \vee (g^L(x) = 0,\ h^L(x) \leq 0), \tag{9}$$

representing L alternative regimes of feasibility. This general disjunction forms the logical core of mixed-integer `DisP` and underlies all subsequent examples and algorithms.

9.2 ML and RL Enhancements

Differentiable Neural Logic for Disjunctive Reasoning. This paragraph introduces the `dNL` formulation, defining continuous conjunction and disjunction operators f_{conj} and f_{disj}, the initialization of logic-layer parameters ($w_i, c, x_i \in [0, 1]$), and the differentiable logical loss L_{logic}. These constructs enable smooth gradient-based reasoning over symbolic disjunctions. This concept is later employed in Algorithm 9 (S0_9–S3_9) for differentiable reasoning, reused in Algorithm 10 (S0_{10}, S2_{10}, S3_{10}) for neural-assisted logical updates, and referenced in Algorithm 3 (S2_3, S3_3) for backpropagation through logical constraints.

The differentiable logical operators are taken from the `dNL` formulation introduced in Section 3. Algorithm 9 summarizes their integration in learning disjunctive constraints.

Algorithm 9 demonstrates how differentiable neural logic integrates symbolic disjunctions into continuous neural optimization. In (S0$_9$), the logical layer parameters w_i, temperature coefficient c, and predicate activations $x_i \in [0,1]$ are initialized to represent soft truth values. In (S1$_9$), differentiable conjunction and disjunction functions $f_{\text{conj}}(x)$ and $f_{\text{disj}}(x)$ are computed using smooth sigmoid activations $m_i = \sigma(cw_i)$, providing continuous approximations of Boolean logic. During (S2$_9$), the model evaluates a symbolic or reinforcement-learning–based loss L_{logic} and performs backpropagation through the logical operators to refine the weights w_i. Finally, (S3$_9$) checks for convergence, halting when the loss or logical consistency stabilizes. Overall, Algorithm 9 enables NNs to reason over disjunctive relations in a differentiable manner, combining interpretability from logic with adaptability from learning.

Algorithm 9 Differentiable neural logic learning for disjunctive reasoning

```
Initialization
(S0$_9$) Initialize logic layer parameters $w_i$, temperature $c > 0$, and predicate activations $x_i \in [0,1]$.
repeat
    Forward computation
    (S1$_9$) Compute differentiable conjunction and disjunction $f_{\text{conj}}(x)$ and $f_{\text{disj}}(x)$ (defined by (18) in Appendix B.1).
    Learning step
    (S2$_9$) Evaluate symbolic or RL-based loss $L_{\text{logic}}$. Backpropagate through logical operators to update weights $w_i$.
    Convergence check
    (S3$_9$) If loss $L_{\text{logic}}$ stabilizes or logical accuracy converges, stop.
until convergence of differentiable logic representation
```

Figure 8 illustrates how differentiable logic layers allow neural models to learn and reason over smooth logical disjunctions.

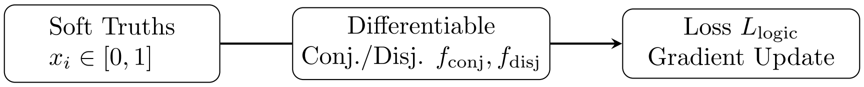


Figure 8: Differentiable neural logic (dNL): continuous conjunction/disjunction operators enable gradient-based optimization over logical constraints.

Together, these learning-enhanced disjunctive frameworks demonstrate how neural architectures and RL can approximate, adapt, and optimize within complex disjunctive structures—bridging symbolic reasoning and data–driven decision-making.

9.3 Synthesis: DisP Meets Learning

This paragraph 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.

Table 6: Representative ML/RL Enhancements in DisP (DisP).

Problem Domain	**ML/RL Contribution within Disjunctive Models**	**Reference**
Job-shop scheduling	GNN + RL for sequential decisions on disjunctive graphs; transferable scheduling policies.	[40]
Job-shop scheduling	Attention-based deep RL with disjunctive graph embeddings; scalable and high-quality solutions.	[16]
Chemical production scheduling	Distributional RL handling precedence/disjunctive constraints; risk-sensitive optimization (CVaR).	[38]
Neuro-symbolic RL	Inductive logic programming integrated with RL; interpretable disjunctive decision rules.	[15]
Program analysis	Data–driven disjunctive modeling for selective context sensitivity; disjunction as an ML bias.	[26]

Detailed mathematical formulations, optimization objectives, and graphical illustrations for the representative domains listed in Table 6 are provided in Appendix D.2.

9.4 First-Order and Logical Constraint Satisfaction Problems (FOL-CSPs)

This subsection defines FOL-CSPs, which generalize classical SAT/CSP by allowing quantifiers and predicates over logical languages $\mathcal{L} = (\mathcal{F}, \mathcal{P})$, interpreted under models $\mathcal{M}$. It establishes quantified and relational reasoning beyond propositional logic. This concept is directly employed in Algorithm 10 ($\mathrm{S0}_{10}$–$\mathrm{S3}_{10}$) to construct and process neural-assisted FOL constraints and connects to

Algorithm 1 ($S1_1$) and Algorithm 2 ($S1_2$) through grounding and propositional reduction.

While Boolean SAT solvers address propositional reasoning, many real-world domains require quantified and relational reasoning beyond propositional clauses. Typical examples include knowledge bases, planning with quantified preconditions, and constraint templates with parameters or relations between objects. These problems are naturally expressed as FOL-CSPs, which extend the classical triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ by defining constraints in a logical language $\mathcal{L} = (\mathcal{F}, \mathcal{P})$ with function and predicate symbols.

A constraint $c_j \in \mathcal{C}$ takes the general form

$$c_j(x_1, \dots, x_k) \;\equiv\; Q_1 y_1 \dots Q_r y_r \;\psi_j(x_1, \dots, x_k, y_1, \dots, y_r),$$

where each $Q_i \in \{\forall, \exists\}$ is a quantifier and ψ_j is a quantifier-free conjunction or disjunction of atomic predicates over variables $x_1, \dots, y_r$. A **model** $\mathcal{M} = (\mathcal{U}, I)$ provides a universe $\mathcal{U}$ and an interpretation I assigning meaning to each predicate and function symbol in $\mathcal{L}$. A solution is an assignment $\mathcal{A} : \mathcal{X} \to \mathcal{U}$ such that $\mathcal{M} \models c_j[\mathcal{A}]$ for all $c_j \in \mathcal{C}$.

Definition 9.1 (FOL-CSP Instance)**.** *A First-Order Constraint Satisfaction Problem is a tuple $(\mathcal{L}, \mathcal{M}, \mathcal{X}, \mathcal{C})$, where $\mathcal{L}$ is a logical language, $\mathcal{M}$ its interpretation structure, $\mathcal{X}$ a finite set of logical variables, and $\mathcal{C}$ a set of first-order constraints over $\mathcal{L}$. A solution $\mathcal{A}$ satisfies $(\mathcal{L}, \mathcal{M}, \mathcal{X}, \mathcal{C})$ iff $\mathcal{M} \models c_j[\mathcal{A}]$ for all $c_j \in \mathcal{C}$.*

Grounding and Reduction. To connect with the propositional SAT/CSP framework, FOL-CSPs are grounded by instantiating quantifiers over finite domains, yielding large but finite propositional subproblems. Formally, each quantified constraint

$$Q_1 y_1 \dots Q_r y_r \;\psi_j(x_1, \dots, x_k, y_1, \dots, y_r)$$

is converted into a finite set of propositional clauses

$$\{\, \psi_j(x_1, \dots, x_k, a_1, \dots, a_r) \mid a_i \in \mathcal{U} \,\},$$

where $\mathcal{U}$ is the domain of instantiation. Learned clause priors and predicate embeddings guide this process by ranking likely instantiations, thus reducing combinatorial explosion and improving solver efficiency. This concept is applied in Algorithm 10 ($S1_{10}$) for clause prioritization and grounding selection, and in Algorithm 3 ($S0_3$) for CNF feature extraction before propagation.

Neural Assistance. This paragraph introduces neural predicate encoders such as GNNs and Transformers, which produce clause embeddings $\phi(\mathcal{K}, q)$ and predict satisfiability probabilities $p_\theta(q|\mathcal{K})$. This mechanism is implemented in Algorithm 10 (S0$_{10}$, S1$_{10}$) to rank quantified clauses.

Recent neuro-symbolic solvers [43] represent predicates, constants, and relations as nodes in a relational graph and employ GNNs or Transformer encoders to predict the relevance or satisfiability probability of each clause. Given a knowledge base $\mathcal{K}$ and a query q, a neural model estimates the entailment likelihood

$$p_\theta(q \mid \mathcal{K}) \;=\; \sigma\big(\mathbf{w}^\top \phi(\mathcal{K}, q)\big),$$

where $\phi(\mathcal{K}, q)$ is an embedding of the symbolic–graph pair, $\mathbf{w}$ and θ are trainable parameters, and $\sigma(\cdot)$ is the logistic activation. The resulting probabilities serve as priorities for clause selection and grounding order.

Integration with the Unified ML–CSP Framework. This paragraph links neural models with classical LCG/CDCL solvers, describing a feedback loop where learned clause relevance guides grounding and solver conflicts update neural encoders. This integration is realized in Algorithm 10 (S0$_{10}$–S3$_{10}$) and further operationalized in Algorithm 5 (S3$_5$) for feedback-driven clause evaluation.

Within the overall framework, FOL-CSPs extend the ML-enhanced LCG/CDCL loop: (1) neural relevance models rank quantified clauses before grounding, (2) grounding produces propositional constraints used by the standard solver, and (3) feedback from solver conflicts updates the neural encoders. This establishes a recursive interaction between symbolic deduction and statistical learning.

The operation of this neuro-symbolic FOL-CSP module is summarized in Algorithm 10. Each step mirrors the solver phase style (S0$_{10}$–S3$_{10}$) used throughout the paper.

This paragraph summarizes the hierarchical solver structure corresponding to Algorithm 10 (S0$_{10}$–S3$_{10}$): (S0$_{10}$) relational graph construction, (S1$_{10}$) clause relevance estimation, (S2$_{10}$) symbolic–neural unification, and (S3$_{10}$) integration with propositional solvers.

Connection to the Global Framework. This paragraph explains the integrative role of the FOL-CSP module in the overall architecture, corresponding conceptually to Algorithm 3 (S0$_3$). It describes how the FOL-CSP layer bridges propositional reasoning at the SAT level and dynamic optimization at the planning level through ML–RL interaction, establishing a unified constraint-solving pipeline. Formally, the integration preserves a hierarchical dependency:

$$\text{Symbolic Reasoning} \;\rightarrow\; \text{FOL-CSP} \;\rightarrow\; \text{RL}/\texttt{DynP}.$$

By embedding symbolic inference within this ML–RL hierarchy, the framework provides a continuous flow of information from logical clauses to sequential decision policies, supporting consistent reasoning, planning, and scheduling under uncertainty.

Algorithm 10 **Neural-Assisted** `FOL-CSP` **Inference**

Initialization

(S0$_{10}$) Construct the relational graph $\mathcal{G} = (V_{\text{var}} \cup V_{\text{pred}}, E)$ from $\mathcal{M}$ or knowledge base $\mathcal{K}$. Initialize a neural encoder ϕ_θ that produces predicate and clause embeddings, and a relevance predictor $p_\theta(c_j)$ estimating the usefulness of each constraint.

Clause Prioritization

(S1$_{10}$) Rank constraints by their predicted relevance $p_\theta(c_j)$. Select the top-K clauses or those with $p_\theta(c_j) > \tau$ for grounding. Generate partial propositional encodings for these high-confidence clauses.

Guided Symbolic Search

(S2$_{10}$) Perform unification and substitution among selected clauses. Neural similarity between predicate embeddings guides the order of resolution. Conflicts or failed unifications are recorded to refine ϕ_θ.

Integration with Propositional Solvers

(S3$_{10}$) The grounded propositional subset $\mathcal{C}_{\text{grounded}}$ is forwarded to the ML-enhanced `LCG/CDCL` solver. Backpropagated signals from solver conflicts update neural parameters θ, closing the loop between symbolic and statistical reasoning.

Termination

The procedure terminates when all clauses are either satisfied or grounded. If all constraints are satisfied under some assignment $\mathcal{A}$, return `SAT`; otherwise return `UNSAT`.

10 Empirical Evidence from the Literature

To strengthen the theoretical framework introduced in this paper, this section summarizes and analyzes empirical findings reported in prior studies on machine learning–assisted constraint solving and combinatorial optimization. These works collectively demonstrate that data-driven and hybrid symbolic–learning approaches can substantially enhance solver efficiency, scalability, and adaptability.

10.1 RL for Search Heuristics

RL has been repeatedly shown to improve the efficiency of combinatorial and scheduling solvers. For example, Park et al. [40] and Liu et al. [34] combined GNNs with deep RL policies to learn dispatching rules for the job-shop scheduling problem, achieving faster convergence and reduced makespan compared to traditional heuristics. Chen et al. [16] proposed an attention-based RL framework with disjunctive graph embeddings that reduced tardiness and outperformed metaheuristic baselines. In chemical production scheduling, Mowbray et al. [38] used distributional RL to optimize batch sequencing under uncertainty, showing robust performance against domain-specific solvers. These results confirm that RL-guided heuristic selection can effectively replace handcrafted rules in dynamic and stochastic scheduling domains.

10.2 ML for `SAT` and `CSP` Solvers

A substantial body of empirical work demonstrates that machine learning can significantly improve `SAT` and `CSP` solver performance. Liang's PhD thesis [32] and follow-up works such as Audemard and Simon [5], Liang et al. [33], and Bergin et al. [10] showed that learned branching heuristics and clause-quality predictors yield faster convergence on benchmark families. Selsam et al. [49] introduced the `NeuroSAT` architecture, which learns unsupervised message-passing embeddings to predict satisfiability, achieving competitive accuracy across standard `SATLIB`[1] and handcrafted datasets. Subsequent transformer-based extensions [50, 51] further improved scalability and runtime performance on industrial `SAT` instances. Matos et al. [35] successfully applied a hybrid `SAT` and ML pipeline to periodic timetabling, reducing solution time and improving feasibility rates on public transport data. Together, these studies empirically validate that data-driven branching, clause learning, and variable selection can improve search efficiency in Boolean and finite-domain constraint satisfaction problems.

[1] http://www.cs.ubc.ca/~hoos/SATLIB/

10.3 Neural and Hybrid Optimization Frameworks

Several works demonstrate the integration of neural architectures with symbolic optimization models. Oh et al. [26] employed a ML-based disjunctive model for static program analysis, achieving higher accuracy and reduced false positives relative to purely symbolic approaches. Recent advances in physics-inspired and graph-based neural solvers further validate this paradigm: Krutský et al. [30] binarized physics-inspired GNNs for large-scale combinatorial optimization, achieving competitive accuracy with substantially reduced model size. The empirical evaluations in these studies demonstrate that learned neural representations can effectively approximate or complement the symbolic reasoning process underlying classical solvers.

10.4 Summary of Reported Improvements

Table 7 consolidates representative empirical results from the literature that support the feasibility of ML- and RL-assisted constraint solving. The reported improvements consistently indicate significant reductions in search effort and runtime compared with traditional handcrafted heuristics.

Table 7: Representative empirical results supporting ML-assisted constraint solving.

Reference	**Summary of Empirical Findings**
Park et al. (2021) [40]	**Approach:** GNN + RL scheduling policy **Problem Domain:** Job-shop scheduling **Reported Improvement:** 10–25% makespan reduction
Liu et al. (2023) [34]	**Approach:** Deep RL with GNN features **Problem Domain:** Dynamic JSSP **Reported Improvement:** 20–35% runtime improvement
Chen et al. (2023) [16]	**Approach:** Attention-based RL + disjunctive graph **Problem Domain:** Job-shop scheduling **Reported Improvement:** Reduced tardiness by 30%

Continued on next page

Table 7 (continued)

Reference	**Summary of Empirical Findings**
Liang et al. (2016) [33]	**Approach:** ML-guided branching heuristic **Problem Domain:** `SAT` solving **Reported Improvement:** Faster solving on industrial benchmarks
Selsam et al. (2018) [49]	**Approach:** Message-passing GNN (`NeuroSAT`) **Problem Domain:** `SAT`, `CSP` classification **Reported Improvement:** Accurate satisfiability prediction
Matos et al. (2021) [35]	**Approach:** ML + `SAT` hybrid optimization **Problem Domain:** Timetabling **Reported Improvement:** Shorter runtime, higher feasibility
Oh et al. (2019) [26]	**Approach:** ML-enhanced disjunctive model **Problem Domain:** Program analysis **Reported Improvement:** Improved precision, fewer false alarms
Krutský et al. (2025) [30]	**Approach:** Binarized physics-inspired GNN **Problem Domain:** Combinatorial optimization **Reported Improvement:** Comparable accuracy, lower compute cost

10.5 Discussion

The empirical results surveyed above consistently confirm the effectiveness of ML and RL techniques when integrated with constraint programming and combinatorial optimization. Learned heuristics and neural graph representations enable solvers to generalize across problem instances, reduce branching depth, and improve solution quality. These findings provide solid empirical grounding for the unified framework proposed in this paper, reinforcing the claim that hybrid intelligent systems combining symbolic reasoning with data-driven inference are both feasible and performance-enhancing.

11 Current Challenges and Future Directions

Building on the empirical evidence summarized in Section 10, we now examine the remaining challenges and research directions that define the frontier of learning–augmented constraint optimization. The convergence of ML, RL, and classical optimization—including SAT, CSP, GNN-based graph models, and DisP—has transformed automated reasoning and combinatorial problem solving. Yet despite this progress, the integration of symbolic and statistical methods remains far from complete; many hybrid solvers succeed on narrow benchmarks but still struggle with scale, interpretability, and theoretical guarantees.

Goal and intuition. The goal of this section is to synthesize the major challenges that limit current learning–augmented optimization frameworks and to outline research directions that can bridge symbolic logic and adaptive learning more effectively. The intuition is that achieving true autonomy in reasoning systems requires more than incremental improvements in accuracy or runtime—it demands a principled understanding of how knowledge, learning, and inference interact within unified solver architectures.

Motivation. By identifying these open problems, we aim to frame a research agenda for the next generation of hybrid solvers. The discussion that follows highlights both the theoretical and practical frontiers of the field—scalability, interpretability, data efficiency, and reproducibility—each representing an essential step toward explainable, adaptive, and verifiable reasoning systems.

Scalability and Generalization. A fundamental limitation of current learning–based solvers is their difficulty in scaling to industrial-scale instances and generalizing across problem families. GNNs, Transformer architectures, and neural policies often exhibit excellent performance on fixed distributions of benchmark instances but degrade significantly on out-of-distribution or structurally distinct problems. This challenge arises from overfitting to training graphs or clause distributions and from the exponential growth of combinatorial search spaces. Promising research directions include meta-learning and transfer-learning strategies that adapt across domains, as well as hierarchical or modular GNNs that capture compositional structure in large-scale graphs without sacrificing tractability.

Interpretability and Theoretical Guarantees. The opacity of neural reasoning remains a major obstacle for adoption in safety–critical or verifiable applications. Unlike classical CDCL, LCG, or DisP frameworks, where each inference step is logically justified, learning–based solvers provide limited interpretability and few theoretical guarantees of soundness or completeness. Bridging this gap requires (i) the design of explainable architectures combining symbolic constraints with differentiable logic (cf. Algorithm 9), and (ii) the development of formal bounds on generalization, convergence, and suboptimality within learned branching or policy mechanisms. Recent hybrid symbolic–neural approaches that embed continuous relaxations of disjunctions or quantified predicates show promise toward transparent and verifiable ML-enhanced solvers.

Data Efficiency and Self-Supervision. Supervised learning for `SAT`/`CSP` or graph optimization requires large sets of solved instances or optimal solutions, which are costly to obtain. Even with reinforcement-based exploration, reward sparsity and slow convergence hinder efficient policy learning. Future research should focus on self-supervised and unsupervised formulations where constraint satisfaction or logical consistency provides intrinsic training signals. Possible strategies include differentiable constraint penalties (cf. Section 5), curriculum generation of progressively harder instances, and generative models that synthesize diverse and challenging training data for solver generalization.

Integration of ML with Classical Heuristics. Although ML-guided heuristics such as backbone prediction, learning-rate branching, and neural restart scheduling have demonstrated clear runtime benefits (see Algorithms 3–5), their coupling with deterministic solver cores remains ad hoc. A rigorous unification of neural policies with classical heuristic control laws could be achieved by formalizing propagation, branching, and backjumping as differentiable operators embedded within a stochastic control framework. Such formulations would enable joint optimization of solver parameters and learned policies under unified gradient-based objectives, maintaining completeness while improving adaptability.

Hybrid Symbolic–Neural Solvers. The most promising direction lies in integrating symbolic reasoning (logical consistency, inference rules) with neural approximations (pattern recognition, generalization). In `DisP` and `FOL-CSP` frameworks, this synergy is achieved through differentiable logical layers and neural predicate embeddings (Algorithms 9 and 10), yet a complete hybrid pipeline capable of continuous logical inference and discrete optimization remains an open goal. Future solvers should combine (i) symbolic correctness guarantees, (ii) neural adaptability, and (iii) dynamic uncertainty management, potentially via RL-based policy hierarchies or variational logic layers.

Benchmarking and Reproducibility. Comparative evaluation of learning–based solvers remains inconsistent due to heterogeneous datasets, differing instance distributions, and diverse hardware environments. Standardized benchmarking—encompassing not only runtime but also learning overhead, sample efficiency, and cross-domain transfer—is essential to ensure reproducibility and fair comparison. Community-driven initiatives, analogous to SATCOMP[2] and the MiniZinc challenge[3], could be extended to include ML-enhanced and neural–symbolic solvers, establishing a unified experimental baseline.

Cross-Domain Integration. Extending ML-enhanced solvers to broader domains such as hybrid optimization, control, and symbolic reasoning under uncertainty constitutes another key frontier. Incorporating disjunctive and relational structures within continuous optimization (e.g., MIP/NLP) and dynamic control (e.g., DynP and RL) can enable unified solvers that reason over both logical and numerical domains. Potential applications include dynamic resource allocation, explainable planning, neuro-symbolic verification, and adaptive multi-agent systems.

The above challenges define the current research frontier for ML-enhanced optimization.

Outlook. The convergence of symbolic optimization, logical reasoning, and statistical learning defines the emerging paradigm of *learning–augmented reasoning*. Realizing its full potential will require scalable graph representations, interpretable hybrid architectures, and principled theories linking learning to logical inference. Future research should focus on developing general frameworks where deductive structure and data–driven adaptability coexist, leading toward self-improving, explainable solvers that integrate the strengths of mathematics, logic, and machine intelligence.

Reflection on Research Questions. The discussion above provides perspective on the three guiding research questions introduced in Subsection 1.3.

Q1 (Integration of ML and RL into Solver Phases): Significant progress has been achieved in embedding learning components within each phase of LCG/CDCL architectures—propagation, decision-making, conflict resolution, and restart control—through Algorithms 3–5. Nevertheless, integration remains partial: most systems treat learned modules as auxiliary heuristics rather than as fully unified, differentiable operators within the solver loop.

Q2 (Algorithmic and Representational Frameworks): GNN-based architectures, differentiable logic (Algorithm 9), and RL-driven heuristic adaptation have established promising representational paradigms. However, theoretical understanding of these frameworks—particularly in terms of convergence,

[2] SATCOMP is the annual international competition evaluating SAT solvers on standardized benchmarks.

[3] The MiniZinc challenge is an annual international competition comparing constraint solving technologies on standardized MiniZinc problem sets. It focuses on finite-domain propagation solvers and aims to build a library of benchmark models for evaluating solver performance and optimization quality [1].

completeness, and generalization—remains limited. Bridging formal guarantees with neural expressivity is a key goal for future hybrid symbolic–neural solvers.

Q3 (Open Challenges and Research Trends): The major challenges identified—scalability, interpretability, data efficiency, and benchmarking—outline an evolving research frontier. Future work should pursue general theories of learning–augmented reasoning that unify symbolic inference and statistical generalization, supported by reproducible empirical benchmarks.

These reflections connect the survey's synthesis back to its foundational research questions, clarifying both the progress achieved and the opportunities that remain for next-generation learning-enhanced constraint solvers.

Transition to Conclusion. The challenges and opportunities outlined above underscore the continuing evolution of learning–based reasoning frameworks. They motivate the synthesis presented in the following Conclusion, where the modular algorithms developed throughout this paper are unified into a coherent ML/RL–enhanced `LCG`/`CDCL` architecture. Together, these contributions illustrate how statistical learning and symbolic optimization can converge toward adaptive, explainable, and self-improving solver paradigms.

To complement the conceptual discussion above, comprehensive mathematical formulations and additional case-study implementations are provided in the appendices (see Appendices B.1–D.2) for reference and reproducibility.

12 Conclusion

This paper presented an integrated framework that enhances classical `CSP` and `SAT` solvers through embedded ML and RL components. Building upon Algorithm 1 (`LCG`, **Lazy Clause Generation**) and Algorithm 2 (`CDCL`, **Conflict-Driven Clause Learning**) paradigms, four modular extensions were introduced to embed learning–based intelligence directly into each major solver phase.

Together, these extensions form a cohesive learning-augmented solver pipeline that connects symbolic reasoning with adaptive statistical learning.

Algorithm 3 (**ML-Enhanced Propagation**) integrates logistic regression, Monte Carlo backbone prediction, and RL–guided pruning. This enhancement replaces static domain filtering with probabilistic backbone estimation and adaptive consistency enforcement, improving propagation efficiency and reducing early conflicts.

Algorithm 4 (**ML-Enhanced Heuristic Decision**) extends the decision-making process through instance classification, parameter tuning, and neural branching via transformer embeddings. The resulting adaptive policy generalizes across problem families, dynamically aligning branching choices with instance structure and solver performance.

Algorithm 5 (**ML-Enhanced Conflict Analysis and Backjumping**) incorporates predictive clause utility modeling and RL-based restart scheduling, enabling data–driven conflict analysis and more effective backjumping. Neural prioritization techniques such as `NeuroGIFT` further demonstrate improvements in cryptanalytic applications by ranking candidate assignments according to learned structural cues.

Algorithm 6 (**ML Applications Beyond the Core Loop**) extends solver capability through ML-based structural analysis of community-structured `SAT` formulas and interpretable model construction using `SAT`-encoded decision trees. This contribution bridges symbolic reasoning and explainable AI, integrating solver analysis with interpretable model learning.

Collectively, these components form a comprehensive ML/RL-augmented `LCG/CDCL` framework. The integration of statistical inference with symbolic reasoning enhances solver scalability, adaptability, and generalization across structured and real-world domains. Prior empirical and theoretical studies [8, 32, 35, 39, 50, 59] show that learned heuristics consistently outperform static rule-based methods in runtime efficiency, conflict minimization, and decision accuracy.

Overall, this study contributes a step toward autonomous, learning-augmented reasoning systems that unify combinatorial optimization, statistical inference, and symbolic logic within a single, adaptive computational framework. Future research should further formalize the theoretical guarantees of this unified architecture and explore its scalability to real-world, large-scale constraint domains.

Detailed algorithmic derivations, mathematical formulations, and extended examples supporting this framework are included in Appendices A–D, which provide full reproducibility of the presented results.

A Appendix: Classical Algorithms and developed software for `CSPs`

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 $\mathcal{A}$.

A.1 Appendix: Classical Algorithms for CSPs

Arc Consistency and the AC-3 Algorithm. Among propagation methods, the AC-3 procedure (=Algorithm 11) is one of the most widely used for establishing consistency in binary CSPs. It operates by iteratively enforcing consistency on every directed arc (X_i, X_j) in the constraint graph. Whenever a value in the domain of X_i is found inconsistent with all values of X_j, it is pruned, and the neighboring arcs of X_i are reinserted for reconsideration. This process continues until no domain reductions occur. AC-3 guarantees that all binary constraints are locally consistent and terminates with a consistent domain assignment or detects inconsistency if any domain becomes empty.

Algorithm 11 Arc Consistency Algorithm (AC-3)

```
(S0_11) Initialize a queue $Q$ with all arcs $(X_i, X_j)$ in the constraint graph.
repeat
    (S1_11) Remove an arc $(X_i, X_j)$ from $Q$.
    (S2_11) For each value $x \in \mathcal{D}(X_i)$, check whether there exists a value $y \in \mathcal{D}(X_j)$ such that the constraint between $X_i$ and $X_j$ is satisfied. If no such $y$ exists, remove $x$ from $\mathcal{D}(X_i)$.
    (S3_11) If $\mathcal{D}(X_i)$ was reduced, then for each neighbor $X_k$ of $X_i$ (except $X_j$), add $(X_k, X_i)$ to $Q$.
until $Q$ is empty.
(S4_11) If any domain $\mathcal{D}(X_i)$ becomes empty, return inconsistent.
Otherwise, all domains are arc-consistent; return consistent.
```

Backtracking Search (BTS) and Branch-and-Bound (BB). Classical CSP search procedures rely on recursive exploration of the search tree representing partial variable assignments. BTS is the foundational depth-first enumeration scheme for constraint satisfaction. At each recursive call, the current partial assignment $\mathcal{A}_p$ is tested for consistency; if it violates a constraint, the search immediately backtracks (fail); if it assigns all variables consistently, the procedure terminates (success). Otherwise, an unassigned variable x_i is chosen, and the algorithm recursively expands all feasible value choices $v \in \mathcal{D}(x_i)$. This recursive process implicitly enumerates the feasible region defined by the constraints and forms the logical basis of many complete solvers.

Algorithm 12 Depth-First Backtracking Search (tt BTS)

```
(S0$_{12}$) Initialize partial assignment $\mathcal{A}_p = \emptyset$, constraint set $\mathcal{C}$, and variable ordering $(x_1, \ldots, x_n)$.
(S1$_{12}$) If $\mathcal{A}_p$ is total and satisfies all $c_j \in \mathcal{C}$, return success.
If $\exists\, c_j \in \mathcal{C}$ such that $\mathcal{A}_p(\texttt{scope}(c_j)) \notin c_j$, return fail.
(S2$_{12}$) Select the next unassigned variable $x_i$ and iterate over values $v \in \mathcal{D}(x_i)$:
    Add $(x_i, v)$ to $\mathcal{A}_p$ and recursively call DFS($\mathcal{A}_p \cup \{(x_i, v)\}$).
    If a recursive call returns success, propagate upward.
    Otherwise, remove $(x_i, v)$ and try the next $v$.
(S3$_{12}$) If all values in $\mathcal{D}(x_i)$ fail, return fail.
```

The success and fail outcomes thus serve as Boolean indicators of feasibility propagation in the recursion: success marks a complete consistent solution, whereas fail indicates that no consistent extension of the current partial assignment exists. Backtracking ensures completeness but can grow exponentially in the number of variables, motivating the use of cost-based pruning. We discuss the DFS algorithm, below.

When an objective function $f(x)$ is introduced, the same tree structure can be searched using the *branch-and-bound* strategy. Instead of exploring all feasible completions, branch-and-bound computes a lower bound $f_{\min}(\mathcal{A}_p)$ (a valid lower bound on the objective value of all feasible completions of $\mathcal{A}_p$) for each partial assignment. Any node whose bound exceeds the current incumbent f^* can be safely pruned. This converts pure feasibility search into an exact optimization scheme and forms the computational backbone of many mixed-integer and combinatorial solvers.

In summary, Algorithm 12 provides the fundamental logical search mechanism, while Algorithm 13 extends it with numerical bounds to support optimization. Both can be viewed as discrete analogues of recursive dynamic programming, and their decision structure directly parallels the policy search mechanisms used later in reinforcement learning formulations.

Breadth-First vs. Depth-First. Let depth($\mathcal{A}_p$) denote the number of assigned variables. Breadth-first search (BFS=Algorithm 14) explores nodes in increasing depth order, ensuring minimal-depth solutions but with exponential

Algorithm 13 Branch-and-Bound (BB)

```
(S0$_{13}$) Initialize incumbent objective $f^* = +\infty$, root node representing the full feasible domain $\mathcal{D}$, and an empty search stack.
repeat
    (S1$_{13}$) Select a node $\mathcal{A}_p$ from the stack (partial assignment). Compute a valid lower bound $f_{\min}(\mathcal{A}_p)$ and check feasibility of $\mathcal{A}_p$.
    (S2$_{13}$) If $\mathcal{A}_p$ is infeasible, discard the node (fail).
    If $\mathcal{A}_p$ is complete and $f(\mathcal{A}_p) < f^*$, update $f^* := f(\mathcal{A}_p)$ and record $\mathcal{A}^* := \mathcal{A}_p$ (success).
    (S3$_{13}$) Otherwise, if $f_{\min}(\mathcal{A}_p) < f^*$, branch on a variable $x_i$ and push each subproblem $\mathcal{A}_p \cup \{(x_i, v)\}$ for $v \in \mathcal{D}(x_i)$ onto the stack.
    Prune any node with $f_{\min}(\mathcal{A}_p) \geq f^*$.
until stack is empty or optimal solution found.
```

memory requirements. Depth-first search (`DFS`=Algorithm 15) explores along a single branch until conflict, then backtracks. Modern solvers combine both strategies through iterative deepening or restarts, providing full coverage while controlling memory.

Algorithm 14 Breadth-First Search (BFS)

```
(S0$_{14}$) Given constraint set $\mathcal{C}$ over variables $(x_1, \dots, x_n)$, initialize a queue $Q \leftarrow [\mathcal{A}_0]$ with the empty assignment $\mathcal{A}_0 = \emptyset$.
repeat
    (S1$_{14}$) Dequeue the first node $\mathcal{A}_p$ from $Q$.
    If $\mathcal{A}_p$ is total and satisfies all $c_j \in \mathcal{C}$, return success.
    If $\exists\, c_j \in \mathcal{C}$ such that $\mathcal{A}_p(\texttt{scope}(c_j)) \notin c_j$, skip to the next node.
    (S2$_{14}$) For the next unassigned variable $x_i$, generate child nodes $\mathcal{A}'_p = \mathcal{A}_p \cup \{(x_i, v)\}$ for all $v \in \mathcal{D}(x_i)$, and enqueue each $\mathcal{A}'_p$ into $Q$.
until $Q$ is empty.
Return fail.
```

Completeness and Termination. All search algorithms described are complete for finite-domain `CSPs`, guaranteeing termination with either `success` or `fail`. However, in the absence of strong pruning or learned guidance, their expected time complexity remains exponential in $|\mathcal{X}|$, underscoring the importance of heuristic and ML-based acceleration.

Algorithm 15 **Depth-First Search** (DFS)

```
(S0$_{15}$) Given constraint set $\mathcal{C}$ over variables $(x_1, \ldots, x_n)$, initialize a stack
$S \leftarrow [\mathcal{A}_0]$ with the empty assignment $\mathcal{A}_0 = \emptyset$.
repeat
    (S1$_{15}$) Pop the top node $\mathcal{A}_p$ from $S$.
    If $\mathcal{A}_p$ is total and satisfies all $c_j \in \mathcal{C}$, return success.
    If $\exists\, c_j \in \mathcal{C}$ such that $\mathcal{A}_p(\texttt{scope}(c_j)) \notin c_j$, continue.
    (S2$_{15}$) Select the next unassigned variable $x_i$, generate child nodes $\mathcal{A}'_p =$
    $\mathcal{A}_p \cup \{(x_i, v)\}$ for all $v \in \mathcal{D}(x_i)$, and push each $\mathcal{A}'_p$ onto $S$.
until $S$ is empty.
Return fail.
```

From Classical Search to CDCL. Modern SAT solvers extend classical CSP search through conflict-driven clause learning (CDCL), non-chronological backtracking, and restart strategies. These mechanisms augment Algorithm 12 with learned "nogoods" that prevent revisiting inconsistent subspaces. The resulting synergy between propagation, learning, and restart policies provides the conceptual bridge to ML-augmented CSP/SAT solvers discussed in later sections.

Both algorithms enumerate the same search tree but in different traversal orders: BFS guarantees the shallowest solution first but requires large memory, whereas DFS uses minimal memory but may explore deep infeasible paths. Iterative deepening combines the advantages of both by bounding the recursion depth and gradually increasing the limit.

Constraint Graph Representation. Every CSP induces a constraint graph $G = (V, E)$ with $V = \mathcal{X}$ and $(x_i, x_k) \in E$ if variables co-occur in some c_j. Search procedures can then be viewed as traversals of G guided by variable-ordering heuristics. For instance, the *minimum remaining values (MRV)* heuristic selects

$$x_i^* = \underset{x_i \in \mathcal{X} \setminus \mathrm{dom}(\mathcal{A}_p)}{\operatorname{argmin}} |\mathcal{D}(x_i)|$$

to reduce branching factor.

Beyond uninformed strategies such as BFS and DFS, heuristic-guided algorithms use cost estimates to prioritize promising partial assignments.

A* Search. A* is a classical best-first search algorithm combining path-cost accumulation with heuristic guidance. At each step, it expands the node minimizing $f(s) = g(s) + h(s)$, where $g(s)$ is the known cost from the start state and $h(s)$ is a heuristic estimate of the remaining cost to the goal. For each successor s' of state s, $c(s, s')$ denotes the transition cost between them. Under an admissible heuristic, A* is both complete and optimal, serving as the foundation for many classical planning and constraint-search procedures. Algorithm 16 summarizes the standard A* framework.

Algorithm 16 Classical A* Search Algorithm

```
Initialization
(S0_16) Initialize the open list with the start node $s_{\text{start}}$; set $g(s_{\text{start}}) := 0$
and $f(s_{\text{start}}) := h(s_{\text{start}})$. Initialize the closed list as empty.
repeat
    Node Selection and Expansion
    (S1_16) Select from the open list the node $s$ with the smallest $f(s) = g(s) + h(s)$.
    If $s$ is the goal state, terminate and reconstruct the path. Otherwise,
    remove $s$ from the open list and add it to the closed list.
    Successor Generation
    (S2_16) For each successor $s'$ of $s$ with transition cost $c(s, s')$:
        • Compute tentative cost $g'(s') = g(s) + c(s, s')$.
        • If $s' \notin$ open/closed lists or $g'(s') < g(s')$, update:
              $g(s') := g'(s'),\quad f(s') := g(s') + h(s'),$
          and record $s'$'s parent as $s$.
        • Add $s'$ to the open list if not already present.
    Termination
    (S3_16) Repeat (S1_16)–(S2_16) until the open list is empty (no solution) or a
    goal node is found. Return the path of minimal total cost $f(s)$ if successful.
until goal reached or open list empty
```

A.2 Appendix: Software using LCG and CDCL algorithms

Lazy Clause Generation (LCG)-based Solvers. LCG-based solvers integrate the propagation mechanisms of finite-domain constraint programming (CP) with the learning capabilities of Boolean satisfiability (SAT) solvers, combining expressive high-level modeling with efficient conflict-driven search. Among state-of-the-art LCG systems, Chuffed [53] is a solver designed from the ground up around the principles of lazy clause generation. Each finite-domain propagator in Chuffed records the reasons for its propagations as implication clauses, constructing a conflict graph similar to that of CDCL SAT solvers. This enables the derivation of *nogoods*, non-chronological backjumping, and variable activity heuristics based on the VSIDS scheme, achieving substantial reductions in redundant search while maintaining tight integration between the SAT and CP layers.

ECLiPSe [4] provides a foundational constraint logic programming (CLP) environment that has strongly influenced the architecture of modern LCG solvers. Developed as a research and industrial platform for constraint-based modeling, it supports hybrid solvers for integer, real, and Boolean domains, an open module system for defining new propagators, and a robust language interface to external solvers. Its design philosophy—separating modeling, search control, and solver implementation—continues to inform LCG system architecture.

Gecode [47] represents a highly optimized and extensible C++ library for constraint programming. It offers an extensive collection of global constraints, advanced propagation algorithms, customizable search strategies, and efficient multi-core parallel search engines. As an open-source framework under the MIT license, Gecode has become a reference implementation for academic research and a back-end for modeling languages such as MiniZinc. Its modular kernel allows the rapid development and testing of new propagation algorithms and branching heuristics, making it an essential component in the evolution of hybrid CP--SAT technologies.

Finally, Picat [62] is a modern logic-based multi-paradigm language that unifies logic, functional, constraint, and imperative programming. Its built-in constraint modules (cp, sat, mip, and smt) provide a unified interface to multiple solving paradigms, allowing the seamless application of SAT-based encodings within a declarative modeling environment. The PicatSAT compiler integrates LCG-style clause learning with high-level constraint representations, while its tabling and dynamic programming features extend the solver's capabilities to planning and optimization problems. Together, these solvers—from ECLiPSe's CLP foundations to Gecode's extensible architecture, Chuffed's hybrid clause learning, and Picat's declarative multi-paradigm integration—illustrate the continuous convergence of CP and SAT paradigms that defines the modern LCG landscape.

Conflict-Driven Clause Learning (CDCL)-based Solvers. CDCL-based solvers form the core of modern propositional and satisfiability modulo theories (SMT) solving. They extend the classical DPLL procedure with conflict-driven clause learning, non-chronological backjumping, restarts, and variable activity heuristics. Among the most influential solvers in this lineage, MiniSAT [21] stands as a minimalist yet extensible implementation that crystallized the modern CDCL architecture. It introduced efficient data structures such as watched literals for Boolean constraint propagation and offered a clean C++ interface, allowing it to serve as a foundation for later high-performance solvers. Glucose [5], derived from MiniSAT, pioneered the use of the LBD (Literal Blocks Distance) metric to evaluate the quality of learnt clauses, introducing the concept of "glue clauses" that connect fewer decision levels and thus contribute most effectively to pruning the search space. This innovation led to predictive clause management strategies that became standard across subsequent solvers.

PicoSAT [12] emphasized low-level performance optimization through memory-efficient occurrence lists and aggressive restart policies. Its innovations in data layout and proof trace compression improved both runtime and proof generation efficiency, establishing new baselines for solver engineering. MapleSAT [33] integrated machine learning concepts into branching heuristics via the Learning Rate Branching (LRB) algorithm, viewing variable selection as an online optimization process inspired by reinforcement learning. This approach significantly improved solver adaptability across heterogeneous benchmark categories.

Kissat [13], a recent descendant of CaDiCaL, exemplifies a modern high-performance CDCL solver engineered for simplicity, determinism, and cache efficiency. Written in C, it combines optimized restarts, inprocessing, and lightweight preprocessing while maintaining the lean structure of MiniSAT. CryptoMiniSAT [52] extends the CDCL framework to cryptographic reasoning by adding native XOR constraint propagation through Gaussian elimination, enabling efficient treatment of algebraic relations in cryptanalysis problems.

Beyond pure SAT solving, Z3 [19] and Yices [20] extend CDCL into the domain of SMT solving. Z3 integrates Boolean reasoning with first-order theories such as arithmetic and arrays via the DPLL(T) architecture, providing a powerful platform for verification, synthesis, and symbolic reasoning. Similarly, Yices combines a DPLL-based Boolean core with specialized theory solvers for arithmetic, arrays, bit-vectors, and uninterpreted functions, supporting MAX-SMT, unsat-core extraction, and model construction. Both solvers implement flexible APIs and serve as backends in major formal verification environments, including PVS and SAL. Together, these CDCL-based solvers—spanning from MiniSAT's modular foundation to Kissat's streamlined engineering, Glucose's clause-quality learning, MapleSAT's machine-learning heuristics, and Z3/Yices's theory integration—define the state-of-the-art in SAT and SMT solving technology.

B Appendix: Mathematical Details of Learning Methods

This appendix provides additional mathematical formulations and algorithmic details that complement Section 3. It expands on gradient-based learning dynamics, graph neural architectures, attention mechanisms, and reinforcement-learning procedures that form the computational foundation for hybrid symbolic–learning solvers.

B.1 Training Dynamics of Deep NNs

Learning proceeds via *backpropagation*, which applies the chain rule to propagate gradients from output to input layers:

$$\frac{\partial L}{\partial W_\ell} = \frac{\partial L}{\partial h_{\ell+1}} \frac{\partial h_{\ell+1}}{\partial h_\ell} \frac{\partial h_\ell}{\partial W_\ell}, \qquad h_{\ell+1} = \sigma(W_\ell h_\ell + b_\ell).$$

This recursive update mechanism enables end-to-end differentiability across all parameters. In hybrid optimization frameworks, such differentiability permits integrating DNN components with symbolic solvers via gradient-based feedback.

B.2 GNNs and Attention Models

A supervised ML model learns a parametric mapping $f_\theta : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ denotes the input space and $\mathcal{Y}$ the prediction target. The model parameters θ are optimized using the standard supervised learning objective $L(\theta)$ defined in Section 3, typically minimized by stochastic gradient descent (`SGD`) or adaptive methods such as `Adam` [27].

Classification Models. When the prediction target is categorical, ML models often use the softmax mapping to represent class probabilities

$$\Pr(y = k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^\top \mathbf{x} + b_k)}{\sum_{j=1}^K \exp(\mathbf{w}_j^\top \mathbf{x} + b_j)}, \tag{10}$$

where $\mathbf{w}_k \in \mathbb{R}^d$ and $b_k \in \mathbb{R}$ are learned parameters for class k. This model underlies instance classification and solver-selection tasks introduced later.

GNN Representations. For graph-structured inputs $G = (V, E)$, ML models often employ GNNs to capture relational dependencies between variables or constraints. Given node embeddings $h_v^{(k)} \in \mathbb{R}^d$ at layer k, each iteration of message passing is defined by

$$m_v^{(k)} = \sum_{u \in \mathcal{N}(v)} \phi_{\text{msg}}(h_u^{(k)}, e_{uv}), \qquad h_v^{(k+1)} = \phi_{\text{upd}}\left(h_v^{(k)}, m_v^{(k)}\right), \tag{11}$$

for all $v \in V$, where ϕ_{msg}, ϕ_{upd}, and ϕ_{read} are learnable neural functions (typically small MLPs), and e_{uv} encodes edge features such as precedence or resource type (the readout operator ϕ_{read} aggregates node embeddings into a fixed-size vector representation – e.g., mean, sum, or attention pooling – ensuring permutation invariance). A global graph representation is obtained through a permutation-invariant readout,

$$h_G = \phi_{\text{read}}\Big(\{h_v^{(K)} \mid v \in V\}\Big),$$

used as input to downstream decision models. Permutation invariance ensures that the learned graph representation remains consistent under node reordering. Here K denotes the total number of message-passing layers (or iterations) in the GNN, so that $h_v^{(K)}$ is the final embedding of node v after K propagation steps.

`node2vec` Embeddings. The `node2vec` algorithm [24] learns continuous vector representations of nodes in a graph by simulating biased random walks that capture both local and global structure. Formally, let $\mathcal{G} = (V, E)$ be a graph, and let $\mathcal{N}_r(v) = (v_1, v_2, \dots, v_r)$ denote a length-r random walk starting from node $v \in V$. The objective is to learn an embedding function

$$f : V \to \mathbb{R}^d$$

that maximizes the likelihood of preserving network neighborhoods under a skip-gram model:

$$\max_f \sum_{v \in V} \sum_{u \in \mathcal{N}_r(v)} \log \Pr(u \mid f(v)), \;\; \Pr(u \mid f(v)) = \frac{\exp(f(u)^\top f(v))}{\sum_{w \in V} \exp(f(w)^\top f(v))}. \quad (12)$$

Here, $\mathcal{N}_r(v)$ denotes the multiset of nodes visited within a random walk of length r starting from node v, and (p, q) are the return and in–out parameters that control the breadth-first and depth-first exploration biases controlling whether the random walk revisits nearby nodes or explores distant ones. The resulting embeddings $f(v)$ serve as dense feature vectors for each node, used in subsequent graph-based learning tasks.

Attention Mechanisms. Transformer-based architectures generalize GNN aggregation through scaled dot-product attention:

$$\operatorname{attn}(Q, K, V) = \operatorname{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \quad (13)$$

where $Q, K, V \in \mathbb{R}^{n \times d_k}$ are the query, key, and value matrices obtained by linear projections of node features, and d_k is the key embedding dimension. This operator allows learning global dependencies over disjunctive or clause graphs while preserving differentiability.

Let $\mathrm{Enc}_\theta : \mathbb{R}^{n\times d_x} \to \mathbb{R}^{n\times d_z}$ denote the Transformer encoder parameterized by θ, which computes a contextual embedding of the input X:

$$\mathrm{Enc}_\theta(X) = \mathrm{FFN}\big(\mathrm{attn}(Q, K, V)\big), \tag{14}$$

where $\mathrm{attn}(Q, K, V)$ is the multi-head attention output defined in (13). The feed-forward network (FFN) applies non-linear transformations and residual connections, producing contextualized embeddings $\mathrm{Enc}_\theta(X)$ that capture dependencies among all input elements.

Pointer Attention. The pointer attention mechanism extends (13) by using attention weights to select a single element from the encoder outputs rather than computing a weighted sum. At decoding step t, the model assigns a probability distribution over encoder positions:

$$\alpha_{ti} = \frac{\exp\big((W_Q q_t)^\top (W_K x_i)\big)}{\sum_j \exp\big((W_Q q_t)^\top (W_K x_j)\big)}, \qquad \Pr(a_t = i) = \alpha_{ti},$$

where $q_t \in \mathbb{R}^{d_k}$ is the decoder query vector at step t, $x_i \in \mathbb{R}^{d_x}$ is the encoder representation of element i, and $W_Q \in \mathbb{R}^{d_k\times d_q}$ and $W_K \in \mathbb{R}^{d_k\times d_x}$ are learned projection matrices for the query and key transformations. The coefficient α_{ti} denotes the probability of selecting encoder position i, and a_t represents the index of the element chosen at decoding step t. This mechanism, introduced by [58], enables a decoder to “point” directly to variables, operations, or clauses during sequential decision making, a capability later exploited in the attention-based scheduling and branching algorithms.

Logistic Regression. For binary prediction problems, such as polarity estimation in `SAT` solvers, a logistic model computes

$$\Pr(x_i = 1 \mid \mathbf{z}_i) = \frac{1}{1 + e^{-(\mathbf{v}^\top \mathbf{z}_i + c)}}, \tag{15}$$

where $\mathbf{z}_i$ are the feature vectors of variable x_i, $\mathbf{v}$ are learned weights, and c is a bias term. This formulation reappears in the polarity-prediction heuristics described later in this section.

Clause Utility Regression. Clause retention and deletion decisions can be guided by a learned estimate of each clause’s expected utility. Let C denote a learned clause characterized by features such as its size $|C|$, Literal Block Distance $\mathtt{LBD}(C)$, age, and activity score. A regression or neural model predicts a scalar utility value

$$u(C) = f_\theta([\mathtt{LBD}(C),\ |C|,\ \mathrm{age}(C),\ \mathrm{activity}(C)])\,,$$

where f_θ is a parametric function (e.g., a multilayer perceptron) with learnable parameters θ. Clauses with higher predicted utility are prioritized during propagation and conflict analysis, while those with low utility are deprioritized or removed. This formulation provides a unified learning interface for clause management, referenced later in Algorithm 5.

Online Learning and Bandit Models. Some decision processes, such as variable selection or parameter tuning, can be formalized as multi-armed bandit problems. At each round t, the learner chooses an action (or arm) a_t and observes a stochastic reward r_t. A common strategy is the Upper Confidence Bound (`UCB`) policy:

$$a_t = \operatorname*{argmax}_i \left(Q_i + \beta \sqrt{\frac{\ln t}{n_i}} \right),$$

where Q_i is the estimated reward of arm i, n_i counts how many times arm i has been selected, and $\beta > 0$ controls exploration versus exploitation. Action values are updated incrementally by

$$Q_i \leftarrow Q_i + \eta(r_t - Q_i),$$

where η is the learning rate. The objective is to maximize the expected cumulative reward over a time horizon T:

$$\max_\pi \ \mathbb{E}_\pi \left[\sum_{t=0}^{T} r_t \right],$$

where π denotes the stochastic decision policy balancing exploration and exploitation. This framework underlies adaptive heuristics such as learning-rate branching and algorithm configuration used in later sections.

Learning Rate Branching (LRB). A practical instance of bandit-based decision making used in modern `SAT` solvers is the LRB heuristic. Each variable x_i corresponds to an arm with reward $r_t^{(i)}$ inversely proportional to the conflict depth after branching. At iteration t, the policy selects arm i according to

$$\Pr(a_t = i) = \frac{\exp(Q_t(i)/\tau)}{\sum_j \exp(Q_t(j)/\tau)},$$

where $Q_t(i)$ is the estimated utility and $\tau > 0$ controls exploration. Action values are updated incrementally,

$$Q_{t+1}(i) \leftarrow (1-\alpha)Q_t(i) + \alpha r_t^{(i)}.$$

This bandit-based mechanism balances exploration and exploitation in variable selection, serving as a prototype for adaptive branching in `CSP`/`SAT` solvers (detailed later in Algorithm 4).

B.3 Mathematical Foundations of RL

RL formalizes optimization and control as a sequential decision-making process modeled by a Markov Decision Process (MDP)

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

where $\mathcal{S}$ is the set of states, $\mathcal{A}$ the set of actions, $\Pr(s'|s,a)$ the transition probability, $R(s,a)$ the reward, and $\gamma \in (0,1]$ the discount factor. At time t, the agent observes state s_t, selects action a_t, receives reward $r_t = R(s_t, a_t)$, and transitions to s_{t+1}. The goal is to find a policy $\pi_\theta(a|s)$ that maximizes the expected return

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right].$$

For later reference, we define the *Monte Carlo return* as

$$R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k,$$

which represents the discounted cumulative reward collected along a trajectory under policy π_θ.

Policy Gradient Theorem. For a differentiable policy π_θ, the gradient of the expected return is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a)\right],$$

where $Q^{\pi_\theta}(s,a)$ is the true action–value function under policy π_θ, and $V^{\pi_\theta}(s)$ is the corresponding state–value function estimating the expected return from state s. In practical actor–critic implementations, these functions are approximated by NNs $Q_\psi(s,a)$ and $V_\phi(s)$ with parameters ψ and ϕ. The advantage estimate is then computed as

$$\hat{A}_t = Q_\psi(s_t, a_t) - V_\phi(s_t),$$

which measures how much better an action performed compared with the critic's baseline expectation.

Value-Based Methods. In Q-learning, the agent estimates the optimal *action–value function* $Q^*(s,a)$, which satisfies the Bellman optimality equation. The tabular update rule is

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s', a')\big],$$

where α is the learning rate and $\gamma \in (0, 1]$ is the discount factor weighting future rewards. In function-approximation settings such as Deep Q-Networks (DQN), the value function is represented by an NN $Q_\psi(s, a)$ parameterized by ψ. Its parameters are optimized by minimizing the temporal-difference loss

$$L(\psi) = \mathbb{E}\big[(r + \gamma \max_{a'} Q_{\psi^-}(s', a') - Q_\psi(s, a))^2\big],$$

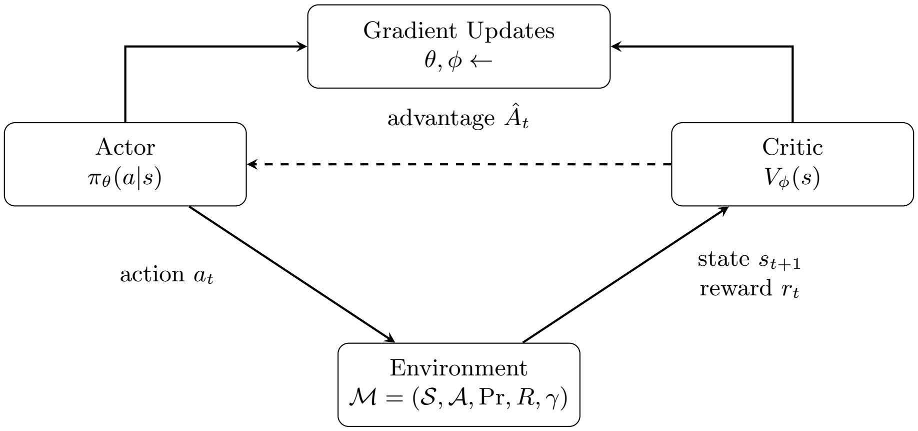
where ψ^- denotes the frozen target-network parameters, and $Q_\psi(s, a)$ serves as an approximation of the true $Q^*(s, a)$ [55].

Algorithm 17 summarizes the `A2C` procedure. In step ($\mathrm{S0}_{17}$), the actor–critic networks and learning rates are initialized. During ($\mathrm{S1}_{17}$), the current policy π_θ interacts with the environment to collect trajectories of states, actions, and rewards. ($\mathrm{S2}_{17}$) computes the discounted returns and advantage estimates $\hat{A}_t$ using the critic's value predictions. The actor parameters are then updated in ($\mathrm{S3}_{17}$) through a policy-gradient ascent step that reinforces actions with positive advantages, while the critic parameters are refined in ($\mathrm{S4}_{17}$) by minimizing the mean-squared error between predicted and observed returns. Finally, ($\mathrm{S5}_{17}$) synchronizes gradients across multiple trajectories to ensure stable, parallel updates.

Figure 9 illustrates how the actor and critic cooperate in `A2C`: the actor generates actions, the critic provides advantage feedback, and both are updated through gradient steps.

Algorithm 17 Advantage Actor–Critic (`A2C`) [36]

```
(S0_17) Initialize actor parameters $\theta$, critic parameters $\phi$, learning rates $(\alpha_\theta, \alpha_\phi)$, and discount factor $\gamma$.
repeat
    (S1_17) Run current policy $\pi_\theta(a|s)$ in the environment for $T$ steps. Collect trajectories $\{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^{T}$ and compute the Monte Carlo return
        $R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k, \qquad r_k = R(s_k, a_k),$
    where $R(s,a)$ is the immediate reward function defined in the MDP, and $R_t$ denotes its discounted cumulative estimate over a sampled trajectory, which is the empirical return computed via Monte Carlo rollout, distinguishing it from the reward function $R(s,a)$ that defines instantaneous feedback.
    (S2_17) Estimate advantages using the critic network:
        $\hat{A}_t = R_t - V_\phi(s_t).$
    (S3_17) Update actor parameters by ascending the policy-gradient objective:
        $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_t|s_t)\, \hat{A}_t.$
    (S4_17) Update critic parameters by minimizing the value loss:
        $\phi \leftarrow \phi - \alpha_\phi \nabla_\phi (V_\phi(s_t) - R_t)^2.$
    (S5_17) Optionally synchronize gradients or parameter averages across multiple parallel trajectories (synchronous update).
until policy convergence or training budget exhausted.
```



Collect trajectories $\{(s_t, a_t, r_t)\}$ and update
actor via policy gradient, critic via value loss.

Figure 9: **Advantage Actor–Critic (`A2C`)** [36]. The actor interacts with the environment to collect trajectories, while the critic evaluates state values to compute the advantage $\hat{A}_t$. Both networks update synchronously to improve the policy.

Algorithm 18 outlines the `PPO` update process. In step (S0$_{18}$), the policy and value networks are initialized together with learning rates, clipping constant ϵ, and discount factor γ. During (S1$_{18}$), the current policy π_θ interacts with the environment to generate trajectories and compute the corresponding discounted returns R_t. Step (S2$_{18}$) estimates advantages $\hat{A}_t$ using the critic's predictions $V_\phi(s_t)$. The policy parameters are then updated in (S3$_{18}$) by maximizing the clipped objective (16), where the ratio $\rho_t(\theta)$ measures the change between new and old policies and the clipping term limits large updates to stabilize learning. Finally, the critic is refined in (S4$_{18}$) by minimizing the value loss $L_V(\phi) = \mathbb{E}_t[(V_\phi(s_t) - R_t)^2]$. The procedure repeats until the policy converges or the training budget is exhausted.

Risk-Sensitive Extensions. In distributional RL, the return from state–action pair (s, a) is treated as a random variable rather than a scalar expectation. Let $Z^\pi(s, a)$ denote the *return distribution under policy* π, defined as the distribution of discounted cumulative rewards obtained by following π after taking action a in state s. The stochastic Bellman operator governing its evolution is

$$\mathcal{T}^\pi Z(s, a) = R(s, a) + \gamma Z^\pi(s', a'),$$

where $R(s, a)$ is the random immediate reward, $Z(s, a)$ represents a generic (estimated) return variable, and $\gamma \in (0, 1]$ is the discount factor. A risk-sensitive

Algorithm 18 Proximal Policy Optimization (PPO) [46]

```
(S0_18) Initialize policy parameters $\theta$, value function parameters $\phi$, clipping constant $\epsilon > 0$, learning rates $(\alpha_\theta, \alpha_\phi)$, and discount factor $\gamma$.
repeat
    (S1_18) Run policy $\pi_\theta$ in the environment for $T$ steps, collect transitions $(s_t, a_t, r_t, s_{t+1})$ and compute discounted returns $R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$.
    (S2_18) Estimate advantages $\hat{A}_t = R_t - V_\phi(s_t)$.
    (S3_18) Compute the probability ratio $\rho_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$, where $\pi_{\theta_{\text{old}}}$ denotes the policy before the current update. Update $\theta$ by maximizing the clipped objective:
        $L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(\rho_t(\theta)\hat{A}_t,\ \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\Big].$   (16)
    The clipping term stabilizes learning by preventing excessively large policy updates that could degrade performance.
    (S4_18) Update $\phi$ by minimizing $L_V(\phi) = \mathbb{E}_t[(V_\phi(s_t) - R_t)^2]$.
until policy convergence or training budget exhausted.
```

objective can be formulated using the Conditional Value-at-Risk (CVaR),

$$\texttt{CVaR}_\alpha(Z) = \mathbb{E}[Z \mid Z \leq F_Z^{-1}(\alpha)], \tag{17}$$

where $\alpha \in (0,1)$ specifies the risk level. The CVaR quantifies the expected loss (or return) in the worst α-fraction of cases, providing a principled measure of tail risk widely used in robust and safety-critical learning [56]. In risk-averse optimization, the objective is typically to minimize $\texttt{CVaR}_\alpha(Z)$, thereby reducing the expected loss in the worst α-fraction of outcomes, rather than maximizing the mean return.

Figure 10 visualizes the PPO framework, where policy updates are constrained by a clipping term to ensure stable, monotonic improvement of the actor network.

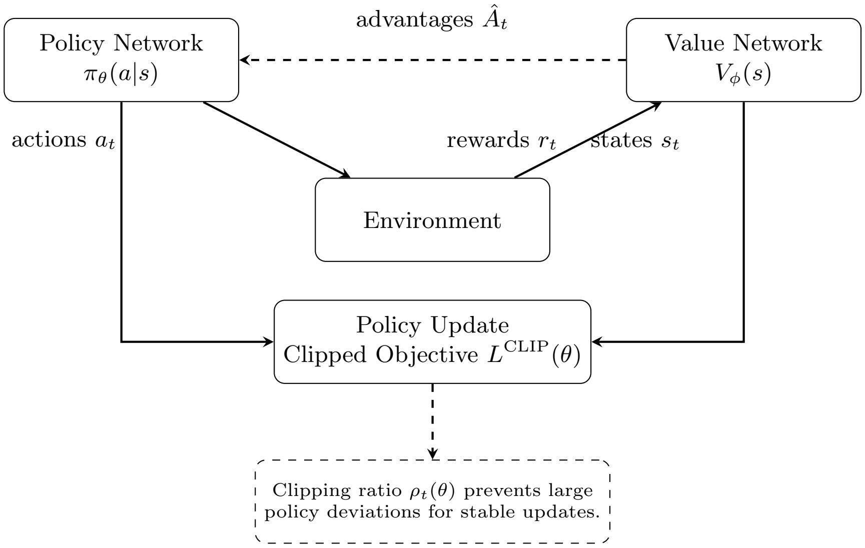


Figure 10: Structure of the `PPO` algorithm. The actor and critic networks generate trajectories, compute advantages, and perform gradient updates on the clipped objective $L^{\text{CLIP}}(\theta)$ to stabilize learning.

Algorithm Configuration. Many solvers expose continuous or discrete parameters $\boldsymbol{\theta} \in \Theta$ that influence performance. The algorithm-configuration problem seeks

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{\phi\sim\mathcal{D}_{\text{inst}}}[T(\phi, \boldsymbol{\theta})],$$

where $T(\phi, \boldsymbol{\theta})$ is the runtime on instance ϕ, and $\mathcal{D}_{\text{inst}}$ denotes the distribution of instances. This optimization viewpoint links ML model selection with solver parameter tuning, discussed later. Note that $\mathcal{D}_{\text{inst}}$ here denotes an instance distribution, distinct from the training dataset $\mathcal{D}_{\text{train}}$ used in supervised learning earlier.

Reward Shaping and Training Loop. In optimization-oriented RL, the reward r_t often reflects the improvement of an objective function over time. If J_t denotes the current objective value, a natural shaping scheme is

$$r_t = J_{t-1} - J_t,$$

so that positive rewards correspond to reductions in the objective (i.e., performance improvement). Each training iteration alternates between data collection under the current policy, advantage estimation, and parameter updates using Algorithm 18. This iterative loop constitutes the backbone of RL integration in subsequent sections.

B.4 Differentiable Neural Logic and Notation

Differentiable Neural Logic(dNL) provides a continuous counterpart to symbolic reasoning, enabling smooth logical operations within gradient-based training. Logical relations are represented through differentiable operators:

$$f_{\text{conj}}(x) = \prod_i (1 - m_i(1 - x_i)), \qquad f_{\text{disj}}(x) = 1 - \prod_i (1 - m_i x_i), \tag{18}$$

where $x_i \in [0, 1]$ denotes a soft truth value, and $m_i = \sigma(cw_i) \in [0, 1]$ is a learned weighting coefficient obtained via a sigmoid activation with temperature parameter $c > 0$. These functions provide smooth relaxations of Boolean conjunction ($\wedge$) and disjunction ($\vee$), thereby enabling continuous optimization over logical expressions. The temperature coefficient c controls the smoothness of the logical transition: larger values of c make the soft logic closer to crisp Boolean behavior. These differentiable operators form the foundation of neural–symbolic reasoning frameworks for continuous logical optimization.

This section establishes the ML and RL terminology, notation, and algorithms that are repeatedly referenced in the forthcoming integrations with DisP and SAT solving frameworks.

C Appendix: Network Flow and Graph Optimization Formulations

This appendix provides full mathematical formulations for the classical network-flow and graph-optimization problems discussed in Sections 4 and 8. Each model is presented with its objective, constraints, and variable definitions, together with optional diagrams illustrating graph structure.

C.1 Network Flow and Disjunctive Optimization Models

Traffic Routing Optimization (TRO). The TRO problem [41] with quality-of-service (QoS) guarantees aims to determine optimal routing strategies that minimize overall network latency or maximize throughput while satisfying QoS constraints such as bandwidth, delay, and jitter.

Let the network be represented by a directed graph $G = (V, L)$, where V is the set of nodes and L is the set of directed links (or edges). Each link $l \in L$ has an associated latency (or delay) cost function $c_l(x_l)$ that depends on the flow x_l traversing it, and a capacity limit $u_l > 0$ specifying the maximum allowable flow. Denote by P the set of all admissible paths in the network, where $P_l \subseteq P$

is the subset of paths that include link l, and by $P_{\text{in}}(v)$ and $P_{\text{out}}(v)$ the sets of paths entering and exiting node $v \in V$, respectively. The goal is to minimize the total network cost:

$$\min_x \sum_{l \in L} c_l(x_l),$$

subject to flow conservation and link capacity constraints.

The TRO problem can be formulated as:

$$\begin{aligned} \min \quad & \sum_{l \in L} c_l(x_l) \\ \text{s.t.} \quad & \sum_{p \in P_{\text{in}}(v)} x_p - \sum_{p \in P_{\text{out}}(v)} x_p = 0, && \forall v \in V, \\ & \sum_{p \in P_l} x_p \leq u_l, && \forall l \in L, \\ & x_p \in s_p \mathbb{Z}_+, && \forall p \in P, \end{aligned} \tag{19}$$

where x_l is the total flow on link l, x_p is the flow assigned to path p, $c_l(x_l)$ is a nonlinear latency or congestion cost function (often convex and increasing), u_l denotes the capacity of link l, and s_p represents a flow scaling parameter associated with discrete routing units. The first constraint enforces **flow conservation**, ensuring that the inflow equals the outflow at each intermediate node, and the second enforces **capacity constraints** limiting the flow on each link.

Traffic routing optimization is fundamental in modern operations research and telecommunications. It is widely applied in Internet traffic engineering, intelligent transportation systems, and large-scale communication networks, where maintaining QoS guarantees (e.g., latency bounds or bandwidth reservations) is critical. An illustrative network example is shown in Figure 11.

Wireless Network Spectrum Allocation (WNSA). The WNSA problem [61] aims to assign available frequency bands to transmitters in a wireless network in order to minimize overall interference while satisfying channel availability and interference threshold constraints.

Let T denote the set of wireless transmitters (e.g., base stations, access points, or other transmitting devices) and F denote the set of available frequency bands (channels). Each transmitter $i \in T$ must be assigned exactly one frequency $f \in F$. When two transmitters $i, j \in T$ are assigned frequencies that cause overlapping signals or nearby spectral leakage, they generate interference quantified by a nonlinear function $I_{ij}(x_i, x_j)$. The objective is to minimize the total interference across all transmitter pairs while maintaining acceptable signal quality:

$$\min_x \sum_{i,j \in T} I_{ij}(x_i, x_j).$$

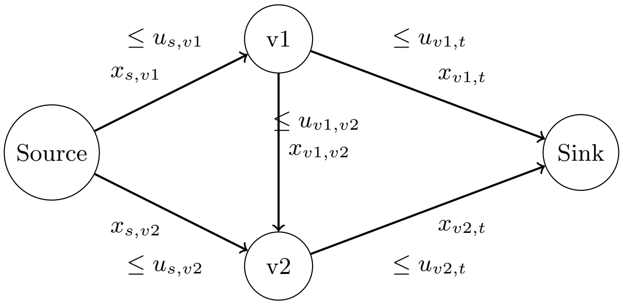


Figure 11: Traffic routing optimization: flows x_l travel from the source to the sink through intermediate nodes v_1 and v_2. Flow conservation ensures that inflow equals outflow at each intermediate node, while link capacities u_l limit total traffic. The objective minimizes overall latency $\sum_{l \in L} c_l(x_l)$ under QoS constraints.

The WNSA problem can be formulated as the following mixed-integer nonlinear program:

$$\begin{aligned} \min \quad & \sum_{i,j \in T} I_{ij}(x_i, x_j) & \\ \text{s.t.} \quad & \sum_{f \in F} x_{if} = 1, && \forall i \in T, \\ & I_{ij}(x_i, x_j) \leq \delta, && \forall i, j \in T, \\ & x_{if} \in \{0, 1\}, && \forall i \in T,\, f \in F, \end{aligned} \tag{20}$$

where $x_{if} = 1$ if transmitter i is assigned frequency f, and 0 otherwise; $I_{ij}(x_i, x_j)$ is a nonlinear interference function measuring signal overlap or channel conflict between transmitters i and j; $\delta > 0$ is the allowable interference threshold, generally $\delta \in (0, \infty)$. The first constraint ensures that each transmitter is assigned exactly one frequency band, while the second restricts the interference between any two transmitters to remain below the threshold δ.

If channel interference is symmetric, then $I_{ij}(x_i, x_j) = I_{ji}(x_j, x_i)$. The parameter $\delta > 0$ denotes the permissible interference threshold, typically determined by system QoS limits.

The WNSA problem is fundamental in wireless communications and network optimization. Applications include dynamic spectrum management in cognitive radio networks, frequency assignment in cellular and Wi-Fi systems, and interference-aware resource scheduling in 5G and next-generation wireless infrastructures. An illustrative example is shown in Figure 12.

Energy-Efficient Network Design (EEND). The EEND problem aims to design a communication or data network that minimizes total energy consump-

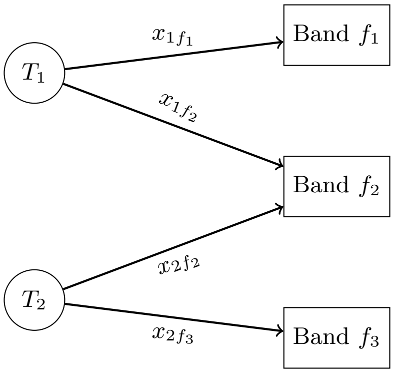


Figure 12: Wireless spectrum allocation: transmitters T_1 and T_2 must each be assigned one frequency band $f \in F$. Assignment variables x_{if} indicate chosen bands, while interference constraints $I_{ij}(x_i, x_j) \le \delta$ limit simultaneous use of conflicting channels.

tion by activating only a subset of available links and devices, while maintaining network connectivity and satisfying all traffic demands.

Let $G = (V, L)$ be a directed or undirected graph where V denotes the set of network nodes and L the set of candidate links. Each link $l \in L$ incurs an energy cost represented by a nonlinear function $E_l(x_l)$, where $x_l \in \{0,1\}$ is a binary variable indicating whether the link l is active ($x_l = 1$) or inactive ($x_l = 0$). The total energy consumption must not exceed the upper limit $\overline{e}$, which represents the available network energy budget ensuring feasibility. Let $\delta(v)$ denote the set of links incident to node v, and let K denote the set of traffic demands, where each d_k specifies the required flow or capacity to be supported in the network. The objective is to select a subset of links that minimizes total energy consumption while ensuring full connectivity and demand satisfaction.

The EEND problem can be formulated as the following mixed-integer nonlinear program:

$$\begin{aligned} \min \quad & \sum_{l \in L} E_l(x_l) \\ \text{s.t.} \quad & \sum_{l \in \delta(v)} x_l \ge 1, && \forall v \in V, \\ & \sum_{l \in L} x_l \ge d_k, && \forall k \in K, \\ & \sum_{l \in L} E_l(x_l) \le \overline{e}, \\ & x_l \in \{0,1\}, && \forall l \in L. \end{aligned} \tag{21}$$

The first constraint ensures that each node in the network has at least one active link, thereby maintaining overall connectivity. The second constraint guarantees that the network can accommodate all required traffic demands d_k. The third constraint limits the total energy consumption to the predefined energy budget $\overline{e}$ (which represents the total available energy budget limiting network operation costs). The binary decision variables x_l activate or deactivate network links, enabling an energy-efficient configuration that balances operational cost and performance.

The EEND problem is a critical problem in sustainable telecommunications and computing infrastructures. It arises in green data center management, smart grid communication systems, and next-generation Internet backbones, where energy savings must be achieved without compromising connectivity or service quality. An illustrative example is presented in Figure 13.

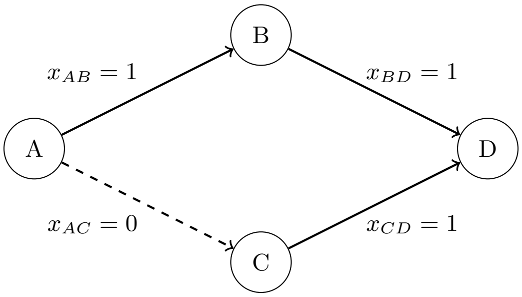


Figure 13: Energy-efficient network design: nodes A–D are connected by links. Binary variables x_l indicate whether a link is active (solid) or inactive (dashed). The objective is to minimize $\sum_l E_l(x_l)$ subject to connectivity and demand constraints.

Load Balancing in Data Centers (LBDC). The LBDC problem [2] seeks to distribute workloads across multiple servers in a way that minimizes total energy consumption while ensuring that capacity and latency constraints are satisfied.

Consider a data center system consisting of a set of servers (or computing nodes) J and a set of workloads or tasks $\mathcal{S}$ to be processed. Each workload $j \in J$ must be allocated among servers according to the system demand d_j. Each server $i \in \mathcal{S}$ has a finite processing capacity u_i and incurs a power consumption $P_i(x_i)$, which is typically a nonlinear increasing function of its utilization x_i. The objective is to determine an allocation of workloads that minimizes the total energy consumed while ensuring demand satisfaction and respecting server capacities.

The LBDC problem can be formulated as the following nonlinear mixed-integer program:

$$\begin{aligned} \min \quad & \sum_{i \in \mathcal{S}} P_i(x_i) \\ \text{s.t.} \quad & \sum_{i \in \mathcal{S}} x_{ij} = d_j, \quad \forall j \in J, \\ & \sum_{j \in J} x_{ij} \leq u_i, \quad \forall i \in \mathcal{S}, \\ & x_i \in s_i \mathbb{Z}_+, \qquad \forall i \in \mathcal{S}, \end{aligned} \tag{22}$$

where x_{ij} denotes the allocation of workload j to server i, and x_i represents the total workload handled by server i. The first constraint ensures that each workload demand d_j is completely satisfied by the aggregate allocation from all servers. The second constraint enforces server capacity limits, guaranteeing that no server operates beyond its capacity u_i. The final constraint defines the discrete or quantized nature of the workload units, where s_i denotes a scaling factor corresponding to the granularity of tasks processed by server i.

The LBDC problem is central to modern cloud computing and distributed computing infrastructures. It underpins energy-aware scheduling, dynamic server provisioning, and adaptive workload management in large-scale data centers. Optimally balancing load across servers reduces both energy costs and operational latency while improving system reliability and sustainability. Empirical studies such as [9] demonstrate the practical impact of energy-aware load balancing in large-scale cloud data centers. By dynamically consolidating workloads and adapting server utilization through heuristic or migration-based strategies, significant reductions in power consumption can be achieved while maintaining quality-of-service guarantees. This aligns closely with the optimization framework in (22), which formalizes workload distribution and capacity constraints as an energy-minimization problem under discrete operational limits. An illustrative example is shown in Figure 14.

Network Security and Intrusion Detection (NSID). The NSID problem [45] seeks to determine the optimal placement of security devices—such as firewalls, intrusion detection systems (IDS), or monitoring sensors—to minimize total deployment cost while ensuring complete coverage of critical network nodes and links.

Let $G = (V, E)$ represent a communication network, where V is the set of nodes (e.g., routers, switches, or servers) that must be protected and E is the set of links connecting them. Let N denote the set of candidate locations where sensors or IDS devices can be deployed. Each device $i \in N$ incurs a deployment cost c_i and provides coverage to a subset of nodes incident to it, denoted by $\delta(v)$. The binary decision variable x_i indicates whether a device is deployed at location i $(x_i = 1)$ or not $(x_i = 0)$. The objective is to minimize the total installation cost while maintaining full network coverage and adhering to a deployment budget γ.

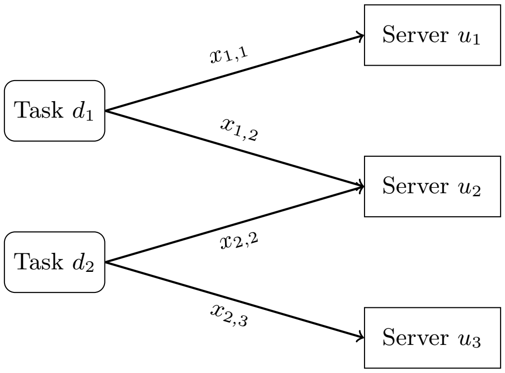


Figure 14: Load balancing in data centers: workloads d_j are distributed among servers with capacity u_i. Variables x_{ij} denote the amount of workload assigned from task j to server i. The objective minimizes total power consumption $\sum_i P_i(x_i)$ subject to demand and capacity constraints.

The NSID problem can be formulated as the following binary integer program:

$$\begin{aligned} \min \quad & \sum_{i \in N} c_i x_i \\ \text{s.t.} \quad & \sum_{j \in \delta(v)} x_j \geq 1, \quad \forall v \in V, \\ & \sum_{i \in N} c_i x_i \leq \gamma, \\ & x_i \in \{0,1\}, \qquad \forall i \in N. \end{aligned} \tag{23}$$

Here, the parameter $\gamma > 0$ denotes the total cost budget for deploying security devices.

The first constraint ensures that every network node $v \in V$ is covered by at least one active security device—this is the **coverage constraint**. The second constraint imposes a total cost limit γ, which represents the available deployment budget or a general resource limitation such as power, storage, or bandwidth. The binary variables x_i determine the selection and placement of devices across the network.

This formulation captures a broad class of problems in cybersecurity and network defense, including optimal placement of firewalls, intrusion detection systems, and monitoring agents. It applies to enterprise networks, cloud infrastructures, and Internet of Things (IoT) environments where resources for protection are limited and efficient coverage is critical. An illustrative example is shown in Figure 15.

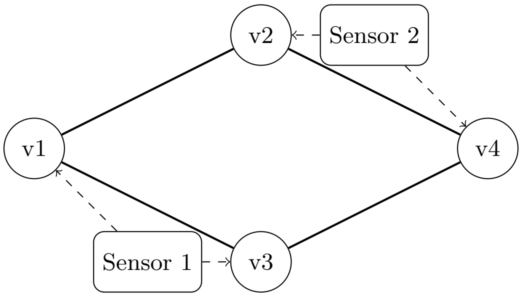


Figure 15: Network security and intrusion detection: binary decision variables x_i determine whether sensors are deployed. Sensor 1 covers nodes v1 and v3, while Sensor 2 covers nodes v2 and v4. The objective minimizes total cost subject to coverage and budget constraints.

Transport Route Selection (TRS). This subsection discusses the TRS problem as a representative example of optimization models that incorporate **logical (disjunctive) constraints**. Two complementary formulations are presented to highlight both the *practical* and *theoretical* aspects of such problems. The first example provides a **complete mixed-integer linear programming (MILP)** model capturing transportation costs, flow conservation, capacity limits, and logical route-selection rules within a realistic network. The second, titled *Disjunctive Representation of Alternative Routes*, abstracts the same idea into a **compact logical form** that isolates the essential disjunction between alternative feasible routes. Together, these examples illustrate how disjunctive conditions arise naturally in transport planning and how they can be modeled either as part of an integrated MILP system or as a standalone logical construct—bridging the gap between **network optimization practice** and the **foundations of DisP**.

Example 1: Transport Route Selection. The TRS problem [25] seeks to determine an optimal set of transportation routes that minimize total shipping cost while satisfying logical, capacity, and supply–demand balance constraints. This problem frequently arises in logistics network design, multimodal transportation planning, and supply chain optimization.

Let $\mathcal{S}$ denote the set of supply nodes, D the set of demand nodes, and R the set of feasible transportation routes. For each route (i, j), the decision variable x_{ij} represents the quantity of goods transported, and the binary variable y_{ij} indicates whether route (i, j) is selected ($y_{ij} = 1$) or not ($y_{ij} = 0$). The unit transportation cost on route (i, j) is denoted by c_{ij}, and u_{ij} denotes its capacity limit. Each node i is associated with a supply or demand quantity b_i, where $b_i > 0$ for suppliers and $b_i < 0$ for consumers. The objective is to minimize the total transportation cost while maintaining network feasibility and logical route consistency.

The TRS problem can be formulated as the following mixed-integer linear program:

$$
\begin{aligned}
\min \quad & \sum_{(i,j)} c_{ij} x_{ij} & & \\
\text{s.t.} \quad & \sum_{j} x_{ij} - \sum_{j} x_{ji} = b_i, & & \forall i \in \mathcal{S} \cup D, \\
& \sum_{(i,j) \in R} y_{ij} \geq 1, & & \\
& y_{ij} + y_{ik} \leq 1, & & \forall i, j, k, \\
& y_{ij} \leq y_{kl}, & & \\
& x_{ij} \leq u_{ij} y_{ij}, & & \forall (i,j), \\
& y_{ij} \in \{0,1\}, & & \forall (i,j), \\
& x_{ij} \geq 0, & & \forall (i,j).
\end{aligned}
\tag{24}
$$

In this formulation, the first constraint enforces **flow conservation**, ensuring that the net flow at each node equals its supply or demand b_i. The second to fourth constraints constitute the **disjunctive constraints**, which ensure logical route selection rules. Specifically, the second constraint guarantees that at least one feasible route is chosen, the third prevents conflicting or parallel routes from being activated simultaneously, and the fourth enforces hierarchical or dependency relationships between routes. The fifth constraint imposes **capacity limits** on each selected route. The binary variables y_{ij} determine which routes are active, while continuous variables x_{ij} capture the flow of goods along those routes.

Transport route selection models are used in multimodal logistics systems, freight network optimization, and supply chain planning. They help determine the most cost-effective configuration of shipping lanes, trucking paths, or rail connections while respecting infrastructure capacity and demand fulfillment. An illustrative example is shown in Figure 16.

Example 2: Disjunctive Representation of Alternative Routes. Consider a transportation network where goods can be shipped through one of several alternative routes. Suppose that if route A is chosen, the flow variables x must satisfy capacity and travel-time constraints $h^1(x) \leq 0$, whereas if route B is chosen, the constraints $h^2(x) \leq 0$ apply. This naturally leads to a disjunctive formulation

$$\big(h^1(x) \leq 0\big) \;\vee\; \big(h^2(x) \leq 0\big),$$

which captures the logical choice between route A and route B. Such modeling is typical in logistics and supply chain optimization, where route alternatives,

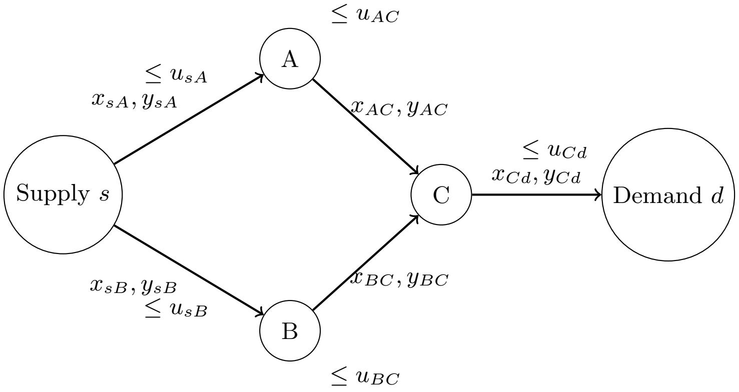


Figure 16: Transport route selection: goods flow from a supply node s to a demand node d through alternative routes (via A or B to C). Decision variables x_{ij} represent transported quantities, while binary variables y_{ij} indicate selected routes. Capacity constraints u_{ij} restrict feasible flows, and disjunctive constraints prevent conflicting or redundant route choices.

congestion thresholds, or contractual restrictions require disjunctive conditions; for example, see Figure 17.

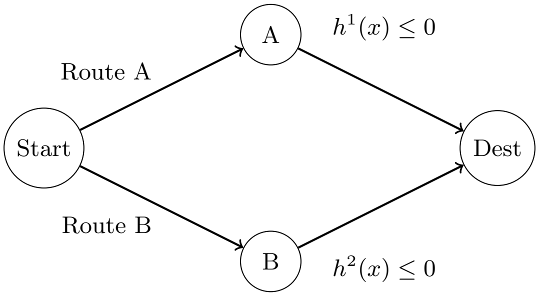


Figure 17: Transport route selection: goods can be sent either via Route A or Route B. Each route is associated with its own capacity and travel-time constraints ($h^1(x) \leq 0$ or $h^2(x) \leq 0$). The disjunction enforces that at least one feasible route must be chosen.

Machine Configuration in Manufacturing (MCM). We here discuss the MCM problem as a canonical example of **disjunctive modeling** in production and scheduling systems. Two complementary formulations are presented to highlight both the *practical* and *theoretical* aspects of disjunctions in manufacturing optimization. The first example introduces a MINLP formulation that integrates task assignment, capacity, precedence, and sequencing constraints within a flexible manufacturing environment. The second, titled *Disjunctive*

Representation of Machine Alternatives, abstracts these ideas into a **logical either–or structure** that isolates the essential technological disjunction between alternative machine configurations. Together, these examples demonstrate how production scheduling can be expressed either as an integrated MILP/MINLP system or as a logical disjunction between feasible machine-specific subsystems—bridging the gap between **industrial scheduling applications** and the `DisP` **framework** introduced by Balas [6].

Example 1: Machine Configuration in Manufacturing. The MCM problem [6] aims to determine the optimal assignment of tasks to machines and their execution sequence in order to minimize total operational cost while satisfying capacity, precedence, and scheduling constraints.

Let T denote the set of manufacturing tasks and M the set of available machines. Each task $i \in T$ must be processed by exactly one machine $j \in M$. The binary variable x_{ij} equals 1 if task i is assigned to machine j and 0 otherwise. The start and completion times of task i are represented by s_i and C_i, respectively. Each task i has a known processing time p_i, and assigning it to machine j incurs a cost d_{ij}. The scheduling of tasks on a shared machine is controlled by precedence and non-overlapping constraints, where h is a sufficiently large positive constant (big-M) used to linearize disjunctive relations. The objective is to minimize the total cost of task–machine assignments while ensuring that each task is processed exactly once, machine capacities are respected, and precedence constraints are maintained.

The MCM problem can be formulated as the following mixed-integer nonlinear program:

$$\begin{aligned}
\min \quad & \sum_{i\in T}\sum_{j\in M} d_{ij}x_{ij} && \\
\text{s.t.} \quad & \sum_{j\in M} x_{ij} = 1, && \forall i \in T, \\
& s_i + p_i \le s_k + h(1 - x_{ij}x_{kj}), && \forall i,k \in T,\ i \ne k,\ j \in M, \\
& C_i \le s_k, && \forall i,k \in T, \\
& C_i = s_i + p_i, && \forall i \in T, \\
& x_{ij} \in \{0,1\}, && \forall i \in T,\ j \in M, \\
& s_i \ge 0, && \forall i \in T, \\
& \text{either } s_i + p_i \le s_k \text{ or } s_k + p_k \le s_i, && \forall i,k \in T,\ i \ne k,\ j \in M.
\end{aligned} \tag{25}$$

The first constraint ensures that each task is assigned to exactly one machine (**task assignment constraint**). The second constraint prevents overlapping tasks on the same machine (**machine capacity constraint**) through a big-M

formulation. The third and fourth constraints define **precedence** and **completion time relations**. The fifth and sixth constraints specify binary and nonnegativity conditions, while the final disjunctive constraint enforces non-overlapping schedules for any pair of tasks assigned to the same machine. Here, $h > 0$ is a sufficiently large constant (Big-M) satisfying $h \gg \max_i p_i$, ensuring that disjunctive scheduling constraints are correctly linearized.

Machine configuration models are fundamental to production planning and manufacturing systems. They are widely applied in job-shop scheduling, flexible manufacturing systems, and robotic cell operations, where efficient task assignment and sequencing directly influence throughput and energy consumption. An illustrative example is presented in Figure 18.

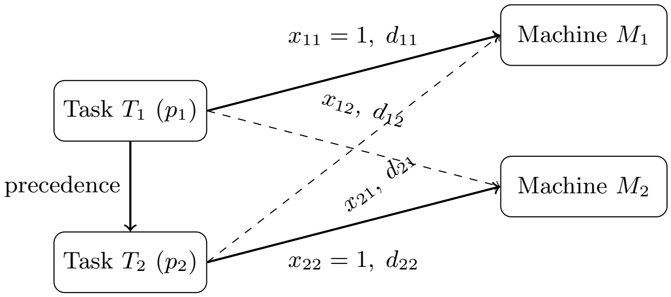


Figure 18: Machine configuration in manufacturing: each task i must be assigned to exactly one machine j, with binary variable x_{ij}. Costs d_{ij} depend on the assignment. Here, Task T_1 is assigned to Machine M_1 and Task T_2 to Machine M_2. Dashed arrows represent alternative assignments. A precedence constraint requires T_1 to finish before T_2 starts.

Example 2: Disjunctive Representation of Machine Alternatives. In a flexible manufacturing system, a product may be processed on one of several machines, each with its own setup cost and operating constraints. For instance, if the product is assigned to machine 1, then equality constraints $g^1(x) = 0$ and inequalities $h^1(x) \leq 0$ enforce the technical requirements of machine 1. If instead machine 2 is chosen, the feasible set is defined by a different system $g^2(x) = 0$, $h^2(x) \leq 0$. The assignment can be written disjunctively as

$$\big(g^1(x) = 0,\ h^1(x) \leq 0\big) \ \vee \ \big(g^2(x) = 0,\ h^2(x) \leq 0\big).$$

This illustrates how disjunctions capture *either-or* technological choices. In practice, the inclusion of integer variables ensures that only one configuration is activated, which is critical in mixed-integer `DisP` (MIDNP); for example, see Figure 19.

The optimization models and graph-based formulations above provide the foundation for integrating data–driven and learning–based heuristics. In the follow-

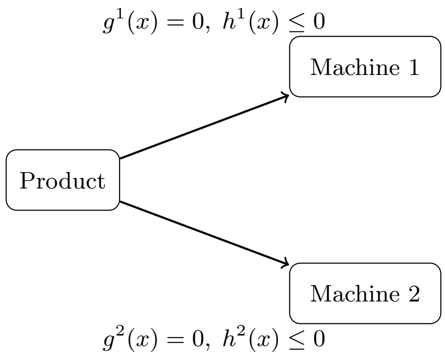


Figure 19: Machine configuration: a product can be assigned either to Machine 1 or Machine 2. Each machine has its own equality and inequality constraints defining feasibility. The disjunctive formulation ensures that exactly one machine configuration is selected.

ing section, we introduce the ML and RL frameworks that will later enhance these classical solvers.

C.2 Classical Graph Optimization Problems

Shortest Path Problem (SPP). The SPP aims to determine the path of minimum total weight between two specified vertices in a weighted graph.

Let $G = (V, E)$ be a directed (or undirected) graph with nonnegative edge weights (or costs) $c_{ij} \geq 0$ for all $(i, j) \in E$. Two distinct vertices $s, t \in V$ are designated as the *source* and *destination* (or *target*) nodes, respectively. The objective is to find a path from s to t that minimizes the total cost:

$$\min_{P \in \mathcal{P}_{s,t}} c(P) = \sum_{(i,j) \in P} c_{ij},$$

where $\mathcal{P}_{s,t}$ denotes the set of all s–t paths in G.

The SPP can equivalently be formulated as a flow-based linear program:

$$\begin{aligned} \min \quad & \sum_{(i,j) \in E} c_{ij} x_{ij} \\ \text{s.t.} \quad & \sum_{\{j \mid (i,j) \in E\}} x_{ij} - \sum_{\{j \mid (j,i) \in E\}} x_{ji} = \begin{cases} 1, & \text{if } i = s, \\ -1, & \text{if } i = t, \\ 0, & \text{otherwise,} \end{cases} \\ & x_{ij} \geq 0, \quad \forall (i,j) \in E, \end{aligned} \tag{26}$$

where the decision variable x_{ij} represents the amount of flow on edge (i, j). In an integer setting, $x_{ij} = 1$ if edge (i, j) lies on the shortest s–t path, and $x_{ij} = 0$ otherwise. The first set of constraints enforces *flow conservation*: one unit of flow is sent from the source s and received at the sink t, ensuring a continuous path from s to t.

The shortest path problem is fundamental in operations research, with applications in transportation and logistics, telecommunication and network routing, and project scheduling and management. An illustrative example is provided in Figure 20.

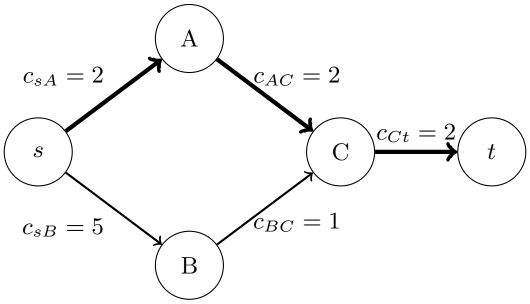


Figure 20: Shortest path problem (SPP): given edge weights c_{ij}, the goal is to find a path from s to t with minimum total cost. Here, the optimal path (thick) is $s \to A \to C \to t$ with cost $2 + 2 + 2 = 6$.

Minimum Spanning Tree (MST). The MST problem seeks a subset of edges that connects all vertices in a connected, weighted graph with the minimum possible total edge cost.

Let $G = (V, E)$ be an undirected and connected graph, where V is the set of vertices and E is the set of edges. Each edge $(i, j) \in E$ is associated with a nonnegative weight (or cost) $c_{ij} \geq 0$. A *spanning tree* of G is a subgraph $T = (V, E_T)$ that connects all vertices of V without forming any cycles. The objective of the MST problem is to find such a spanning tree T that minimizes the total edge cost:

$$\min_{E_T \subseteq E} c(E_T) = \sum_{(i,j) \in E_T} c_{ij}.$$

The MST problem can be expressed as the following integer program:

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in E} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{(i,j)\in E} x_{ij} = |V| - 1, \\
& \sum_{(i,j)\in E(S)} x_{ij} \le |S| - 1, \quad \forall S \subset V,\ S \neq \emptyset, \\
& x_{ij} \in \{0,1\}, \qquad\qquad \forall (i,j) \in E,
\end{aligned}
\tag{27}
$$

where $x_{ij} = 1$ if edge (i,j) is included in the spanning tree and $x_{ij} = 0$ otherwise. The first constraint ensures the correct number of edges in a spanning tree, and the second (subtour elimination) prevents cycles.

The MST problem has numerous applications in operations research and computer science, including cluster analysis in data science, supply chain and transportation network design, and infrastructure planning. An illustrative example is shown in Figure 21.

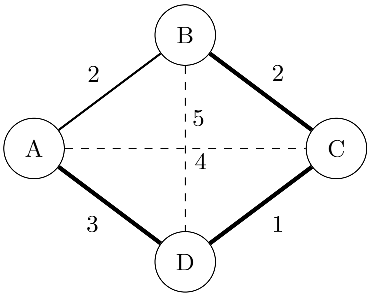


Figure 21: Minimum spanning tree (MST): the selected edges (thick) connect all vertices with minimum total weight $3 + 2 + 1 = 6$.

Maximum Flow Problem (MFP). The MFP seeks to determine the greatest possible amount of flow that can be sent from a designated source vertex to a designated sink vertex through a capacitated network.

Let $G = (V, E)$ be a directed graph where each edge $(i,j) \in E$ has a nonnegative *capacity* $u_{ij} \ge 0$ representing the maximum permissible flow along that edge. Two distinct vertices $s, t \in V$ are designated as the *source* and *sink* (or *target*) nodes, respectively. A *feasible flow* is a function $f : E \to \mathbb{R}_+$ satisfying the following:
1. **Capacity constraints:** $0 \le f_{ij} \le u_{ij}$ for all $(i,j) \in E$;
2. **Flow conservation:** for every vertex $i \in V \setminus \{s,t\}$,

$$\sum_{j:(i,j)\in E} f_{ij} - \sum_{j:(j,i)\in E} f_{ji} = 0.$$

The goal is to maximize the total amount of flow leaving the source (equivalently, entering the sink):

$$\max_f \sum_{j:(s,j)\in E} f_{sj} - \sum_{j:(j,s)\in E} f_{js}.$$

The MFP can be formulated as the following linear program:

$$\begin{aligned}
\max \quad & \sum_{\{j|(s,j)\in E\}} f_{sj} - \sum_{\{j|(j,s)\in E\}} f_{js} \\
\text{s.t.} \quad & \sum_{\{j|(i,j)\in E\}} f_{ij} - \sum_{\{j|(j,i)\in E\}} f_{ji} = 0, \quad && \forall i \in V \setminus \{s,t\}, \\
& 0 \le f_{ij} \le u_{ij}, && \forall (i,j) \in E,
\end{aligned} \tag{28}$$

where f_{ij} denotes the flow on edge (i,j) and u_{ij} its capacity. The objective maximizes the net outflow from the source, subject to flow conservation at all intermediate nodes and edge capacity constraints.

The MFP has broad applications in operations research and network optimization, including transportation and logistics planning, communication and data network design, and resource or workforce assignment. An illustrative example is provided in Figure 22.

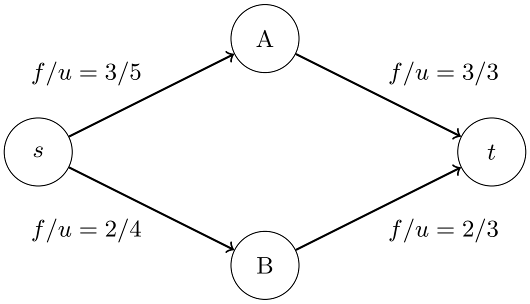


Figure 22: Maximum flow problem (MFP): flows f_{ij} (numerators) are assigned on each edge subject to capacity limits u_{ij} (denominators). The maximum flow from s to t in this example is 5.

Minimum Cut Problem (MCP). The MCP seeks to partition the vertices of a graph into two disjoint sets such that the total weight of the edges crossing the partition is minimized.

Let $G = (V, E)$ be a directed (or undirected) graph with nonnegative edge capacities (or costs) $c_{ij} \ge 0$ for all $(i,j) \in E$. Two distinct vertices $s, t \in V$ are designated as the *source* and *sink* (or target), respectively. The goal is to find a partition (S, T) of V satisfying $s \in S$, $t \in T$, that minimizes the total capacity of edges from S to T:

$$\min_{(S,T)} c(S,T) = \sum_{(i,j)\in E,\, i\in S,\, j\in T} c_{ij}.$$

The minimum cut problem can be expressed equivalently as the following integer program:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in E} c_{ij}x_{ij} \\ \text{s.t.} \quad & x_{ij} \geq y_i - y_j, && \forall (i,j) \in E, \\ & y_s = 1, \quad y_t = 0, \\ & 0 \leq x_{ij} \leq 1, \quad y_i \in \{0,1\}, && \forall i \in V, (i,j) \in E, \end{aligned} \tag{29}$$

where y_i indicates the partition assignment of vertex i ($y_i = 1$ if $i \in S$, $y_i = 0$ if $i \in T$), and x_{ij} is an indicator variable equal to 1 if edge (i,j) crosses the cut from S to T and 0 otherwise.

The minimum cut problem arises in various domains of operations research, including network reliability analysis, supply chain optimization, and project scheduling and resource allocation. An illustrative example is provided in Figure 23.

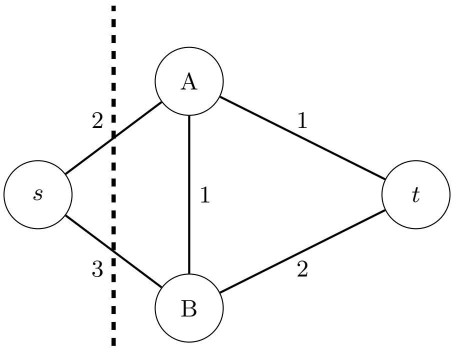


Figure 23: Minimum cut problem (MCP): the dashed line separates $\{s\}$ from $\{A, B, t\}$, cutting edges (s, A) and (s, B) with total cost $2 + 3 = 5$.

Graph Coloring Problem (GCP). The GCP seeks to assign colors to vertices of a graph such that no two adjacent vertices share the same color, while minimizing the total number of colors used.

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. A *proper coloring* of G assigns a color $v \in [k] = \{1, 2, \ldots, k\}$ to each vertex $i \in V$ such that adjacent vertices receive different colors. The objective is to determine the smallest integer k (the *chromatic number*) for which such a coloring exists:

$$\min_{x,\,k} \; k.$$

The GCP can be formulated as the following integer program:

$$\begin{aligned}
\min \quad & k \\
\text{s.t.} \quad & x_{iv} + x_{jv} \leq 1, \quad \forall (i,j) \in E,\ \forall v \in [k], \\
& \sum_{v=1}^{k} x_{iv} = 1, \qquad \forall i \in V, \\
& x_{iv} \in \{0,1\}, \qquad \forall i \in V,\ \forall v \in [k],
\end{aligned} \tag{30}$$

where the binary decision variable x_{iv} equals 1 if vertex i is assigned color v, and 0 otherwise. The first set of constraints enforces that adjacent vertices cannot share the same color, while the second ensures that each vertex receives exactly one color. Minimizing k yields the smallest number of colors needed to properly color the graph.

The graph coloring problem is a classical NP-hard problem with widespread applications in operations research and computer science. Notable applications include register allocation in compiler design, frequency assignment in telecommunications, and task or resource scheduling in manufacturing systems. An illustrative example is shown in Figure 24.

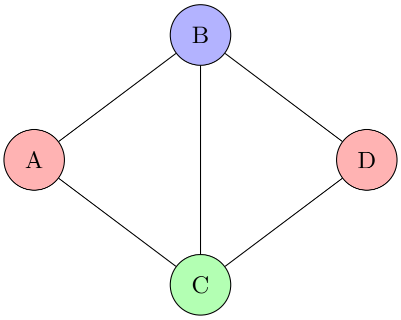


Figure 24: Graph coloring problem (GCP): each vertex is assigned a distinct color such that adjacent vertices differ. Here, three colors (red, blue, and green) suffice to color all vertices.

Traveling Salesman Problem (TSP). The TSP seeks the shortest possible route that visits each vertex exactly once and returns to the starting vertex.

Let $G = (V, E)$ be a complete directed (or undirected) graph where each edge $(i, j) \in E$ is associated with a nonnegative travel cost (or distance) $c_{ij} \geq 0$.

The objective is to determine a Hamiltonian cycle that visits every vertex in V exactly once and minimizes the total travel cost:

$$\min_{x} \sum_{(i,j)\in E} c_{ij} x_{ij}.$$

Here, x_{ij} is a binary decision variable equal to 1 if the salesman travels directly from city i to city j, and 0 otherwise.

The TSP can be expressed as the following integer linear program:

$$\begin{aligned}
\min \quad & \sum_{(i,j)\in E} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{\{j|(i,j)\in E\}} x_{ij} = 1, && \forall i \in V, \\
& \sum_{\{i|(i,j)\in E\}} x_{ij} = 1, && \forall j \in V, \\
& \sum_{(i,j)\in E(S)} x_{ij} \le |S| - 1, && \forall S \subset V,\, S \neq \emptyset, \\
& x_{ij} \in \{0,1\}, && \forall (i,j) \in E,
\end{aligned} \tag{31}$$

where the first two sets of constraints ensure that each city is visited exactly once and departed from exactly once, while the third set (subtour elimination constraints) prevents disconnected cycles (subtours).

The TSP is a cornerstone problem in combinatorial optimization and operations research, with broad applications in logistics and transportation, manufacturing and robotic routing, and scheduling and sequencing. An illustrative example is shown in Figure 25.

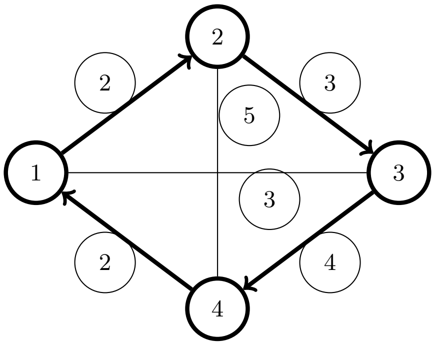


Figure 25: Traveling salesman problem (TSP): a tour visiting all vertices exactly once and returning to the start. The thick cycle represents the optimal route with minimum total cost.

The graph optimization problems discussed—shortest path, spanning tree, flow, cut, coloring, and TSP—form a hierarchy of classical combinatorial problems that underpin many OR applications. They also serve as natural domains for applying ML and RL techniques in subsequent sections.

D Appendix: Extended Examples in `DisP`

D.1 Background on Disjunctive Graphs and Logical Constraints

`DisP` provides a mathematical framework for representing scheduling and sequencing problems in which two operations competing for the same resource cannot overlap in time. Introduced by Balas [6], a *disjunctive graph* $\mathcal{G} = (V, C \cup D)$ encodes a set of operations V, conjunctive arcs C representing precedence constraints, and disjunctive arcs D representing resource conflicts that must be ordered one way or the other.

Classical Formulation. Each operation $i \in V$ has a start time $x_i \in \mathbb{R}_+$ and processing duration $p_i > 0$. For any two operations (i, j) requiring the same machine, exactly one of the following disjunctions must hold:

$$x_i + p_i \leq x_j \quad \text{or} \quad x_j + p_j \leq x_i,$$

ensuring that no two jobs overlap on a shared resource. Precedence relations $(i, j) \in C$ impose additional constraints $x_i + p_i \leq x_j$. The scheduling objective is often to minimize the *makespan*

$$C_{\max} = \max_{i \in V}(x_i + p_i),$$

subject to all conjunctive and disjunctive constraints.

This background establishes the classical and dynamic foundations of disjunctive scheduling on which subsequent sections build, including dynamic GNN–RL integration, risk-sensitive RL, and differentiable neural logic for disjunctive reasoning.

D.2 Learning-Enhanced `DisP`: Extended Case Studies

The following formulations and diagrams illustrate the mathematical structure and ML/RL integration for each domain in Table 6.

(1) Job–Shop Scheduling (GNN + RL on Disjunctive Graphs) [40]. The classical job–shop scheduling problem can be expressed as

$$\begin{aligned}
\min_{x_i,\, C_{\max}} \quad & C_{\max} \\
\text{s.t.} \quad & x_i + p_i \le x_j \;\; \vee \;\; x_j + p_j \le x_i, && \forall (i,j) \in D, \\
& x_i + p_i \le x_j, && \forall (i,j) \in C, \\
& C_{\max} \ge x_i + p_i, && \forall i \in V, \\
& x_i \ge 0,
\end{aligned}$$

where x_i is the start time of operation i, p_i its processing duration, C denotes precedence arcs, and D disjunctive (resource-conflict) arcs. A GNN encodes $\mathcal{G} = (V, C \cup D)$ and an RL agent (e.g., `PPO`) learns a dispatching policy $\pi_\phi(a_t|s_t)$ to minimize the makespan $C_{\max}$.

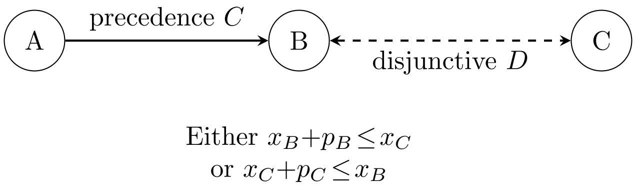


Figure 26: Disjunctive-graph representation of a job–shop scheduling instance.

(2) Job–Shop Scheduling (Attention–based Deep RL) [16]. The attention mechanism captures long-range dependencies among operations:

$$h_i = \mathrm{attn}(Q, K, V)_i = \sum_{j \in V} \alpha_{ij} W_V x_j, \qquad \alpha_{ij} = \frac{\exp((W_Q x_i)^\top (W_K x_j))}{\sum_k \exp((W_Q x_i)^\top (W_K x_k))}.$$

The RL policy $\pi_\theta(a_t|s_t)$ selects the next operation based on attention-weighted embeddings, producing scalable, transferable schedules.

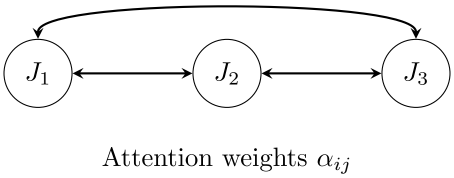


Figure 27: Attention-based encoding of global precedence and disjunction dependencies.

(3) Chemical Production Scheduling (Distributional RL, CVaR) [38]. Let $Z^\pi(s,a)$ denote the random return distribution. A risk-sensitive scheduling problem minimizes expected downside loss:

$$\begin{aligned} \min_{\pi} \quad & \mathtt{CVaR}_\alpha[-Z^\pi(s,a)] \\ \text{s.t.} \quad & x_i + p_i \le x_j \;\; \vee \;\; x_j + p_j \le x_i, \\ & h^l(x) \le 0, \; g^l(x) = 0. \end{aligned}$$

The CVaR objective $\mathtt{CVaR}_\alpha(Z) = \mathbb{E}[Z \,|\, Z \le F_Z^{-1}(\alpha)]$ penalizes the worst α–fraction of outcomes, yielding safer policies for uncertain chemical-production systems.

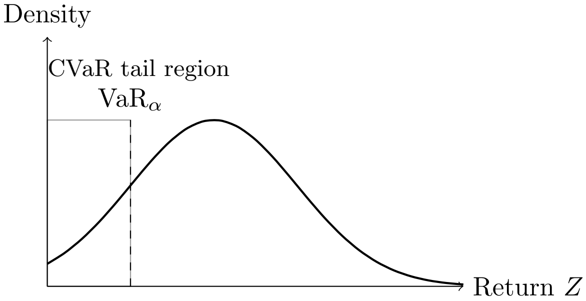


Figure 28: Illustration of the CVaR tail region for risk-sensitive RL scheduling.

(4) Neuro–Symbolic RL (Inductive Logic Programming + RL) [15]. Policies are expressed as differentiable logical combinations:

$$\pi(a|s) = \sigma\left(\sum_l w_l \, f^l_{\text{logic}}(s,a)\right), \qquad f_{\text{disj}}(x) = 1 - \prod_i (1 - \sigma(w_i x_i)).$$

RL optimizes weights w_l to satisfy symbolic rules while preserving differentiability, yielding interpretable disjunctive decision structures.

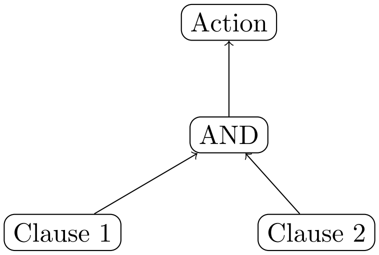


Figure 29: Differentiable logical composition in neuro–symbolic RL.

(5) Program Analysis (Data–Driven Disjunctive Modeling) [26]. Program analysis with selective context sensitivity can be modeled by

$$\bigvee_{l=1}^{L} \big(g_l(x) = 0,\ h_l(x) \le 0\big),$$

where each l denotes an execution context. ML models estimate probabilities $p_l = \Pr(\text{select context } l \mid x)$ to guide which context or abstract domain to analyze.

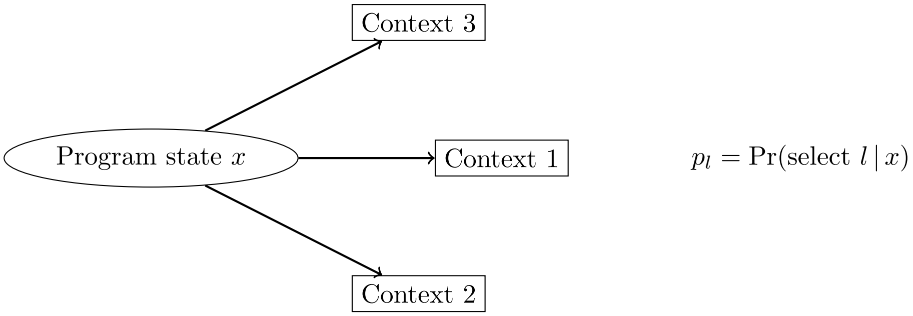


Figure 30: Disjunctive selection of analysis contexts guided by ML probabilities.

Acknowledgements This work received funding from the National Centre for Energy II (TN02000025).

References

[1] The minizinc challenge. https://www.minizinc.org/challenge.html, 2025. Accessed: 2025-10-31.

[2] Muhammad Abdullah Adnan, Ryo Sugihara, and Rajesh K. Gupta. Energy efficient geographical load balancing via dynamic deferral of workload. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 188–195, 2012.

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

[4] Krzysztof R Apt and Mark Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2006.

[5] 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.

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

[7] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications.* Springer Science & Business Media, 2008.

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

[9] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755–768, May 2012.

[10] 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.

[11] 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.

[12] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[13] 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.

[14] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory.* Springer Publishing Company, Incorporated, 2008.

[15] 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.

[16] 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.

[17] 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.

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

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[20] Bruno Dutertre and Leonardo de Moura. The yices smt solver. In *Proceedings of the Satisfiability Modulo Theories Workshop (SMT)*, 2006.

[21] 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.

[22] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. Pmlr, 2017.

[23] Alexander Grigor'yan, Yong Lin, Yuri Muranov, and Shing-Tung Yau. Cohomology of digraphs and (undirected) graphs. *Asian Journal of Mathematics*, 19(5):887–932, 2015.

[24] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864. ACM, August 2016.

[25] Milan Janić. *Advanced transport systems*. Springer, 2014.

[26] 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.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2015.

[28] TN Kipf. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[29] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 4th edition, 2007. A classic book providing a theoretical foundation for combinatorial optimization problems.

[30] Martin Krutskỳ, Gustav Šír, Vyacheslav Kungurtsev, and Georgios Korpas. Binarizing physics-inspired gnns for combinatorial optimization. *arXiv preprint arXiv:2507.13703*, 2025.

[31] Jingyan Li, Yuri Muranov, Jie Wu, and Shing-Tung Yau. On singular homology theories of digraphs and quivers. Technical report, Yanqi Lake Beijing Institute of Mathematical Sciences and Applications (BIMSA), Beijing, China, 2024. Preprint, BIMSA Publication No. 5381.

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

[33] 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.

[34] 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.

[35] 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.

[36] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937. PMLR, 2016. No DOI assigned; official PMLR version.

[37] 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.

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

[39] 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.

[40] 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.

[41] Michael Patriksson. *The Traffic Assignment Problem: Models and Methods*. Dover Publications, revised edition edition, 2015. Covers models and optimization methods for traffic assignment and routing problems.

[42] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, 2022.

[43] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. *Advances in Neural Information Processing Systems*, 30, 2017. NeurIPS 2017.

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

[45] Karen Scarfone and Peter Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. National Institute of Standards and Technology (NIST), 2007. A comprehensive guide to intrusion detection and prevention systems, including network security models and techniques.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[53] 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.

[54] 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.

[55] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

[56] Aviv Tamar, Yonatan Glassner, and Shie Mannor. Optimizing the cvar via sampling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[58] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, pages 2692–2700. Curran Associates, Inc., 2015.

[59] 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.

[60] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[61] Qing Zhao and Behnaam Aazhang. *Dynamic Spectrum Access in Wireless Networks: A Survey*, volume 7 of *Wireless Communications and Mobile Computing*. Springer, 2007. Explores dynamic spectrum allocation in wireless networks, providing models and optimization methods.

[62] Neng-Fa Zhou, Cristian Grozea, Håkan Kjellerstrand, and Oisín Mac Fhearaí. Picat through the lens of advent of code. *arXiv preprint arXiv:2507.11731*, 2025.