Supplemental Material for Machine Learning Algorithms for Improving Exact Classical Solvers in Mixed Integer Continuous Optimization

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

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

Brian Olimba
CEO Olicheza Limited
331 Buruburu Phase 1 Ol Pogoni Road, 76416–00508 Nairobi, Kenya
Email: brian.olimba@olicheza.org

Abstract. This supplemental material includes a collection of Mixed Integer Nonlinear Programming (MINLP) examples, methods, and pseudo-code. It is packaged as `suppMat.pdf` and available at

https://github.com/GS1400/suppMat_ML_MINLP

For completeness, we also include the algorithms illustrating how machine learning and reinforcement learning can enhance MINLP techniques. The original preprint [26] contained ten algorithms in the main text; we have moved them here to keep the submitted version concise. In addition, this supplement provides a brief overview of existing survey papers on machine learning for (M)INLP and `BB` methods, to help situate recent learning-based approaches within the broader literature.

Contents

1 Optimization Problems and Examples

In the main paper [26], the integer and mixed-integer nonlinear optimization problems are defined. However, since we need to refer to them here, it is preferable to restate these definitions.

1.1 Integer Nonlinear Optimization (INLP) problem

In this section, we introduce the standard INLP problem and its variants. Various algorithms to solve these problems are discussed in Section 3 of the main paper [26].

We first define the set of simple bounds

$$\mathbf{X} := \{x \in \mathbb{R}^n \mid \underline{x} \leq x \leq \overline{x}\} \quad \text{with } \underline{x}, \overline{x} \in \mathbb{R}^n \ (-\infty \leq \underline{x}_i < \overline{x}_i \leq \infty \text{ for all } i \in [n]) \tag{1}$$

on variables $x \in \mathbb{R}^n$ (called the **box**). Then, let $[q] := \{1, \cdots, q\}$ for any positive integer value q and formulate the INLP problem as

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

where the real-valued (possibly non-convex) objective function $f : C_{\text{in}} \subseteq \mathbf{X} \to \mathbb{R}$ is defined on the **integer nonlinear feasible set**

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

Here, $s_i > 0$ is a resolution factor (typically $s_i = 1$, corresponding to standard integer variables), the components of the vectors

$$g(x) := \big(g_1, g_2 \ldots, g_m\big) \quad \text{and} \quad h(x) := \big(h_1, h_2 \ldots, h_p\big) \tag{4}$$

are the real-valued (possibly non-convex) constraint functions $g_k : C_{\text{in}} \subseteq \mathbf{X} \to \mathbb{R}$ for $k \in [m]$ and $h_j : C_{\text{in}} \subseteq \mathbf{X} \to \mathbb{R}$ for all $j \in [p]$. If all the functions f, g_k for all $k \in [m]$, and h_j for all $j \in [p]$ are linear, the INLP problem (2) reduces to the integer linear programming (ILP) problem

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & x \in C_{\text{inl}} \end{array} \tag{5}$$

with the **integer linear feasible set**

$$C_{\text{inl}} := \{x \in \mathbf{X} \mid Ax = b, \quad Bx \leq d, \quad x_i \in s_i\mathbb{Z} \ \text{ for all } \ i \in [n]\}, \tag{6}$$

where $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{n \times p}$, $b \in \mathbb{R}^m$, $d \in \mathbb{R}^p$, and $c \in \mathbb{R}^n$.

1.2 Mixed-Integer Nonlinear Programming (MINLP) problem

When additional continuous decision variables are present, the problem becomes a mixed-integer optimization problem. This class of problems is more challenging and requires careful treatment of the interdependence between continuous and discrete decisions. Classical mixed-integer algorithms to solve these problems are discussed in Section 3 of the main paper [26].

We formulate the MINLP problem as

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

where the real-valued (possibly non-convex) function $f : C_{\text{mi}} \subseteq \mathbf{X} \to \mathbb{R}$ is defined on the **mixed-integer nonlinear feasible set**

$$C_{\text{mi}} := \{x \in \mathbf{X} \mid g(x) = 0, \quad h(x) \leq 0, \quad x_i \in s_i\mathbb{Z} \ \text{ for } \ i \in I, \, x_i \in \mathbb{R} \ \text{ for } \ i \in J\}. \tag{8}$$

Here, the box $\mathbf{X}$ comes from (1), I is a subset of $\{1, \cdots, n\}$, $s_i > 0$ is a resolution factor, the components of the vectors (defined by (4)) are the real-valued (possibly non-convex) functions $g_k : C_{\text{mi}} \subseteq \mathbf{X} \to \mathbb{R}$ for $k \in [m]$ and $h_j : C_{\text{mi}} \subseteq \mathbf{X} \to \mathbb{R}$ for all $j \in [p]$. We write x_I and x_J for the subvectors of x indexed by I and J, where $J := [n] \setminus I$. If $I = \emptyset$ and $J = [n]$ are chosen for the MINLP problem (7), then C_{mi} is reduced to the **continuous nonlinear feasible set**

$$C_{\text{co}} := \{x \in \mathbf{X} \mid g(x) = 0, \quad h(x) \leq 0\}; \tag{9}$$

therefore, the MINLP problem (7) is reduced to the continuous nonlinear programming (CNLP) problem

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & x \in C_{\text{co}}. \end{array} \tag{10}$$

If all the functions f, g_k for all $k \in [m]$, and h_j for all $j \in [p]$ are linear, the MINLP problem (7) reduces to the mixed-integer linear programming (MILP) problem

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & x \in C_{\text{mil}} \end{array} \tag{11}$$

with the **mixed-integer linear feasible set**

$$C_{\text{mil}} := \{x \in \mathbf{X} \mid Ax = b, \quad Bx \leq d, \quad x_i \in s_i \mathbb{Z} \ \text{ for all } \ i \in I\}, \tag{12}$$

where $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{n \times p}$, $b \in \mathbb{R}^m$, $d \in \mathbb{R}^p$, and $c \in \mathbb{R}^n$. If $I = \emptyset$ and $J = [n]$ are chosen for the MILP problem (11), then C_{mil} is reduced to the **continuous linear feasible set**

$$C_{\text{col}} := \{x \in \mathbf{X} \mid Ax = b, \quad Bx \leq d\}; \tag{13}$$

hence the MILP problem (11) is reduced to the continuous linear programming (CLP) problem

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & x \in C_{\text{col}}. \end{array} \tag{14}$$

Several MILP examples are provided in Subsection 1.3, including the Crew Scheduling Problem, Knapsack Problem, Vehicle Routing Problem, Facility Location Problem, Energy Grid Optimization, and Hydropower Scheduling.

1.3 Examples of Mixed Integer Optimization Problems

Mixed Integer Optimization Problems (MIPs) are central to a wide array of decision-making tasks where variables may be continuous, integer-valued, or binary, and must satisfy a set of linear or nonlinear constraints. These problems are ubiquitous in industrial, logistical, and engineering applications due to their ability to model discrete decisions under complex operational requirements. Despite their broad applicability, MIPs are often computationally demanding, which has spurred substantial research into both exact and approximate solution methods. To ground our discussion, we now present several illustrative examples of MIPs that capture the modelling challenges and constraint structures commonly encountered in practice. Each example highlights a different application domain, emphasizing the richness and versatility of mixed-integer

formulations. In particular, the **crew scheduling**, **knapsack**, **vehicle routing**, and **facility location** problems are classical **MILP** examples, while the **energy grid optimization** and **hydropower scheduling** problems involve nonlinear physical or power-flow relationships and therefore constitute **MINLP** models.

1.3.1 Crew Scheduling Problem

The **crew scheduling problem** aims to minimize the total cost associated with assigning crew members to shifts while satisfying all workload, demand, skill, and legal and union constraints. This problem can be formulated as

$$\begin{aligned} \min \quad & f(\mathcal{S}) = \sum_{(i,j)\in\mathcal{S}} c_{ij} \\ \text{s.t.} \quad & \mathcal{S} \in \mathcal{F}, \end{aligned}$$

where $\mathcal{S}$ is the subset of crew assignment, c_{ij} is the cost of assigning crew i to shift j, and

$$\mathcal{F} := \{\mathcal{S} \subseteq N \mid \text{workload, demand, skill, and legal and union constraints}\}.$$

Here N contains all potential crew-shift assignments represented as pairs (i, j).

Let us now describe the constraints $\mathcal{F}$. The first constraint is the set of **demand constraints**

$$\sum_{i\in\text{Crew}} x_{ij} \geq d_j, \quad \text{for all } j \in \text{Shifts},$$

where the variable x_{ij} takes the value 1 if a crew member i is assigned to shift j, and 0 otherwise, and d_j is the minimum number of crew members required for a shift j. The second constraint is the set of **workload constraints**

$$\sum_{j\in\text{Shifts}} x_{ij} \leq L_i, \quad \text{for all } i \in \text{Crew},$$

where L_i is the maximum number of shifts that a crew member i can work. The crew member i lacks the skills for the shift j in the **skill** matching constraints $x_{ij} = 0$ for all (i, j), which is the third constraint. The fourth constraint is the **legal and union constraint**

$$x_{ij} + x_{i(j+1)} \leq 1, \quad \text{for all } i \in \text{Crew and } j \in \text{Shifts},$$

where each crew member must have a minimum rest period between consecutive shifts. The working hours for each crew member must not exceed their allowable limits H_i, i.e.,

$$\sum_{j\in\text{Shifts}} w_j x_{ij} \leq H_i, \quad \text{for all } i \in \text{Crew},$$

which is the fifth constraint. Here, w_j denotes the duration of shift j.

1.3.2 Knapsack Problem

The **knapsack problem** aims to maximize the total value of items selected, while ensuring that a capacity constraint on the total weight or volume of the selected items is satisfied. It can be formulated as

$$\begin{aligned} \max \quad & f(\mathcal{S}) = \sum_{i \in \mathcal{S}} v_i \\ \text{s.t.} \quad & \mathcal{S} \in \mathcal{F} := \left\{ \mathcal{S} \subseteq N \mid \sum_{i \in \mathcal{S}} a_i \leq \overline{a} \right\}, \end{aligned} \tag{15}$$

where $\overline{a}$ is the weight capacity of the knapsack and v_i is the value of item i, and a_i is the weight of item i.

1.3.3 Vehicle Routing Problem

The **vehicle routing problem** aims to minimize the total cost of the routes taken by a fleet of vehicles while ensuring that several constraints (**route** and **capacity**) are satisfied. This problem can be formulated as

$$\begin{aligned} \min \quad & f(\mathcal{S}) = \sum_{i,j \in \mathcal{S}} c_{ij} \\ \text{s.t.} \quad & \mathcal{S} \in \mathcal{F} := \{ \mathcal{S} \subseteq E \mid \text{route and capacity constraints} \}, \end{aligned}$$

where $\mathcal{S}$ is the subset of edges E used in the solution and c_{ij} is the cost of traversing the edge (i, j). Let us describe which constraints $\mathcal{F}$ contains.

To begin with, each **decision variable**

$$x_{ij} \in \{0, 1\}, \quad \text{for all } i, j \in \mathcal{N}$$

is binary, indicating whether a vehicle travels directly from customer i to customer j.

The second constraint is the **route coverage** (**demand constraints**)

$$\sum_{i \in \mathcal{N}} x_{ij} = 1, \quad \text{for all } j \in \mathcal{C}, \quad \sum_{j \in \mathcal{N}} x_{ji} = 1, \quad \text{for all } i \in \mathcal{C},$$

where each customer must be visited exactly once and left exactly once by some vehicle. Here, $\mathcal{N}$ is the set of all nodes (including the depot and customers) and $\mathcal{C}$ is the set of customers.

The third constraint is the **subtour elimination constraint**

$$u_i - u_j + (|\mathcal{C}| - 1) \cdot x_{ij} \leq |\mathcal{C}| - 2, \quad \text{for all } i \neq j, \text{ for all } i, j \in \mathcal{C},$$

where auxiliary variables u_i are used to ensure the nonexistence of subtours. The fourth constraint is the **vehicle capacity constraint**

$$\sum_{j \in \mathcal{C}} d_j \cdot x_{ij} \leq Q, \quad \text{for all } i \in \mathcal{V},$$

where each vehicle's total load (the sum of customer demands) must not exceed its capacity Q. Here, $\mathcal{V}$ is the set of vehicles, Q is the capacity of each vehicle, and d_j is the demand of customer j.

1.3.4 Facility Location Problem

The **facility location problem** aims to minimize the total cost consisting of the fixed costs of opening facilities and the transportation costs for assigning customers to facilities while ensuring that facility constraints are satisfied. This problem can be formulated as

$$\begin{aligned} \min \quad & f(\mathcal{S}) = \sum_{i \in \mathcal{S}} c_i + \sum_{j \in C} \min_{i \in \mathcal{S}} d_{ij} \\ \text{s.t.} \quad & \mathcal{S} \in \mathcal{F} = \{\mathcal{S} \subseteq F \mid \text{facility constraints}\}, \end{aligned}$$

where $\mathcal{S}$ is the subset of facilities to open, F is the total set of possible facilities, c_i is the fixed cost of opening the facility i, and d_{ij} is the cost of serving a customer j from the facility i.

Again we use the binary decision variables, where each binary variable $y_i \in \{0, 1\}$ indicates whether the facility i is open or not, and each binary variable $x_{ij} \in \{0, 1\}$ indicates whether a customer j is assigned to the facility i.

Let us describe the constraints $\mathcal{F}$ contains. The first constraint is the **customer assignment constraint**

$$\sum_{i \in F} x_{ij} = 1, \quad \text{for all } j \in \mathcal{C},$$

where each customer $j \in \mathcal{C}$ must be assigned to exactly one facility. The second constraint is the **facility opening constraint**

$$y_i \geq x_{ij}, \quad \text{for all } i \in F,\, j \in \mathcal{C},$$

where a facility i can only serve customers if it is open, which means that if a customer j is assigned to a facility i (i.e., $x_{ij} = 1$), then the facility i must be open (i.e., $y_i = 1$).

1.3.5 Energy Grid Optimization

The **energy grid optimization** problem aims to minimize the total cost, containing both generation and transmission costs, while ensuring that the grid constraints, including power balance, generation limits, transmission limits, power flow relationships, binary facility assignments, and non-negativity, are satisfied. This problem can be formulated as

$$\begin{aligned} \min \quad & f(\mathcal{S}) = \sum_{g \in G} F_g(P_g) + \sum_{l \in L} F_l(P_l) \\ \text{s.t.} \quad & \mathcal{S} \in \mathcal{F} = \{\mathcal{S} \subseteq N \mid \text{power flow and grid constraints}\}, \end{aligned}$$

where N is the set of all nodes (generation plants, consumer nodes, and transmission lines), $G \subseteq N$ is the set of generation nodes (e.g., power plants), $L \subseteq N \times N$ is the set of transmission lines connecting generation plants and consumer nodes, P_g is the power generated at generation node $g \in G$, P_l is the power flowing through transmission line $l \in L$, $F_g(P_g)$ is the generation cost function at plant $g \in G$, and $F_l(P_l)$ is the transmission cost function for line $l \in L$.

Let $x_{gl} \in \{0, 1\}$ for all $g \in G$, $l \in L$ be the set of binary decision variables, where x_{gl} is true if the transmission line l is used for transmitting power from generation plant g, and false otherwise.

Let us describe the constraints that $\mathcal{F}$ contains. The first constraint is the **power balance**

$$\sum_{g \in G} P_g = \sum_{c \in C} D_c + \sum_{l \in L} P_l$$

whose goal is to ensure that the total power generated equals the total power demand plus the power transmitted, where $C \subseteq N$ is the set of consumer nodes (e.g., load centers) and D_c is the power demand at a consumer node $c \in C$. The second constraint is the **generation limits**

$$P_g^{\min} \leq P_g \leq P_g^{\max}, \quad \text{for all } g \in G,$$

whose goal is to ensure that the power generated at each plant is within its specified limits ($P_g^{\min}$ and $P_g^{\max}$). The third constraint is the **transmission limits**

$$P_l^{\min} \leq P_l \leq P_l^{\max}, \quad \text{for all } l \in L,$$

whose goal is to ensure that the power flow through each transmission line is within its specified limits ($P_l^{\min}$ and $P_l^{\max}$). The fourth constraint is the **power flow relationships**

$$P_l = \sum_{g \in G} \alpha_{gl} P_g - \sum_{c \in C} \beta_{lc} D_c, \quad \text{for all } l \in L$$

that express the power flow through each transmission line l in terms of the generation at plant g and the demand at consumer node c. Here α_{gl} denotes the fraction of power generated at node $g \in G$ transmitted through line $l \in L$ and β_{lc} denotes the fraction of demand at consumer node $c \in C$ supplied by transmission line $l \in L$.

The fifth constraint is the **facility allocation constraint**

$$\sum_{g \in G} x_{gl} \leq 1, \quad \text{for all } l \in L$$

whose goal is to ensure that each transmission line is used by at most one generation plant. The sixth constraint is the non-**negativity constraint**

$$P_g \geq 0, \quad P_l \geq 0, \quad \text{for all } g \in G,\, l \in L$$

whose goal is to ensure that power generation P_g and transmission P_l are non-negative.

1.3.6 Hydropower Scheduling

The **hydropower scheduling** problem aims to optimize the operation of a hydropower system over a given planning horizon to minimize total operational costs (or maximize energy generation revenue), while satisfying physical, operational, and environmental constraints. This problem can be formulated as

$$\begin{aligned} \min \quad & f(\mathcal{S}) = \sum_{t \in T} C_t(P_t) \\ \text{s.t.} \quad & \mathcal{S} \in \mathcal{F} = \{\mathcal{S} \subseteq \mathbb{R}^{3|T|} \mid \text{hydrological and operational constraints}\}, \end{aligned}$$

where $\mathcal{S} = (P_t, R_t, S_t)_{t \in T}$, T is the set of periods (e.g., hourly or daily intervals), P_t is the power generated, R_t is the water release, S_t is the reservoir storage at time $t \in T$, and $C_t(P_t)$ is the cost function (or negative revenue function) associated with generation at time t. Here, the decision variables are

$$P_t, R_t, S_t \in \mathbb{R}_{\geq 0}, \quad \text{for all } t \in T.$$

Let us describe the constraints that $\mathcal{F}$ contains. The first constraint is the **reservoir dynamics**

$$S_{t+1} = S_t + I_t - R_t, \quad \text{for all } t \in T,$$

whose goal is to ensure conservation of water volume in the reservoir system, where I_t denotes the natural inflow at time $t \in T$. The second constraint is the **generation relationship**

$$P_t = \eta \cdot R_t \cdot H_t, \quad \text{for all } t \in T,$$

which relates the generated power to the water release and hydraulic head H_t, where η is the conversion efficiency. The third constraint is the **storage limits**

$$S_t^{\min} \leq S_t \leq S_t^{\max}, \quad \text{for all } t \in T,$$

whose goal is to ensure that the water level remains within physically feasible limits. The fourth constraint is the **release limits**

$$R_t^{\min} \leq R_t \leq R_t^{\max}, \quad \text{for all } t \in T,$$

whose goal is to enforce operational and technical bounds on the water release. The fifth constraint is the **environmental flow constraint**

$$R_t \geq R_t^{\text{eco}}, \quad \text{for all } t \in T,$$

whose goal is to ensure that a minimum ecological flow is maintained downstream of the dam. The sixth constraint is the **non-negativity constraint**

$$P_t \geq 0, \quad \text{for all } t \in T,$$

whose goal is to ensure that power generation values are non-negative.

2 Detailed Materials Supporting Branch and Bound Algorithms

This section compiles all detailed algorithms, figures, proofs, and solver descriptions that were summarized in Section 3 of the main paper [26]. The main text presents a concise conceptual and mathematical overview, while the full derivations and illustrations are moved here for completeness.

Subsection 2.1 presents all flowcharts illustrating the core components of classical branch-and-bound (`BB`) algorithms, including node selection, branching, and bounding logic. Subsection 2.2 expands the algorithmic steps and procedural details that underpin the high-level formulations introduced earlier. Subsection 2.2 details the cutting-plane, column-generation, and feasibility-pump mechanisms used to enhance bounding performance. Subsection 2.3 describes key MINLP solvers and their architectures, while Subsection 2.4 provides the theoretical convergence guarantees and analytical remarks. Finally, Subsection 2.5 summarizes solver performance and comparative results across benchmark problems.

2.1 Flowcharts of Classical `BB` Components

Figures 1–9 summarize the standard flow of the classical components used throughout the main paper [26]. Figures 1–3 show the three canonical node-selection rules (best-bound, depth-first, breadth-first). Figures 4 and 5 illustrate

the evolution of feasible regions and the associated branching set. Figures 6 and 7 describe the construction of active sets and the refinement of relaxed subproblems. Figure 8 presents the bound-update logic for continuous and integer relaxations, and Figure 9 provides the main steps of the `iFP` routine. These diagrams are included solely to give a compact visual representation of the classical mechanisms referenced in later sections.

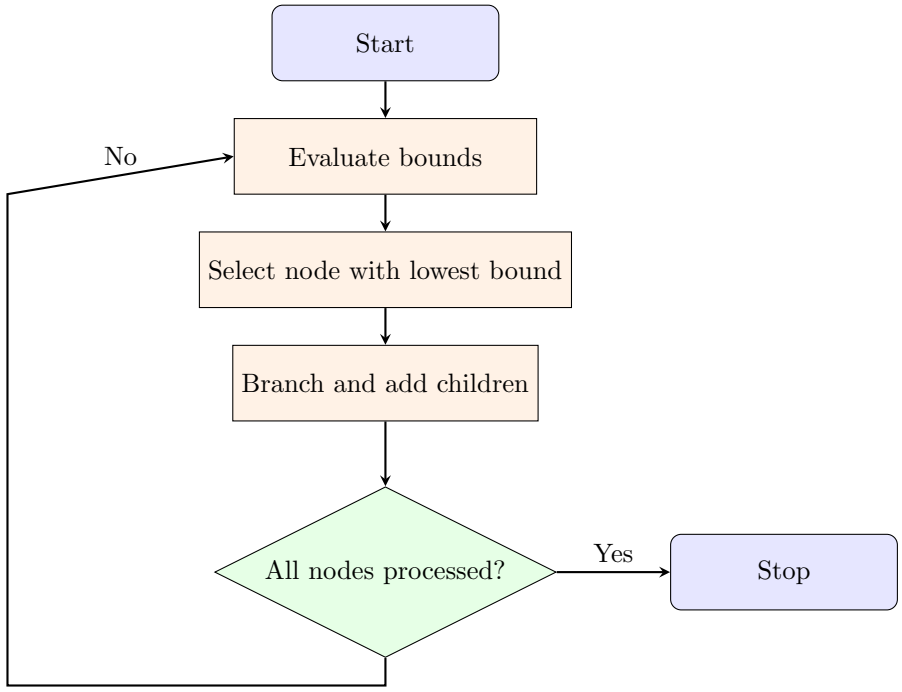

Figure 1: Best-Bound Search (BBS): Selects the node with the lowest lower bound from the active pool. This is a best-first strategy.

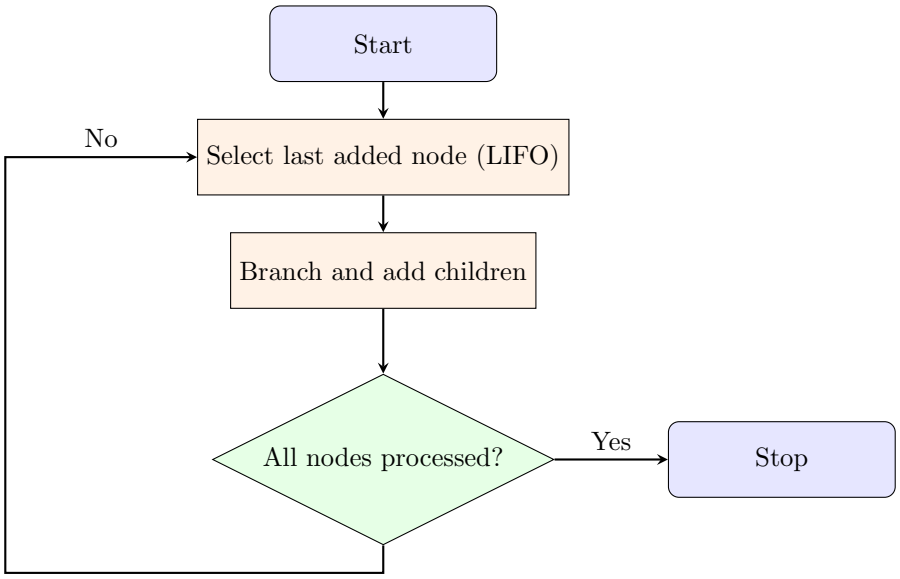

Figure 2: Depth-First Search (DFS): Selects the most recently added node using a Last-In, First-Out (LIFO) stack. This explores one path deeply before backtracking.

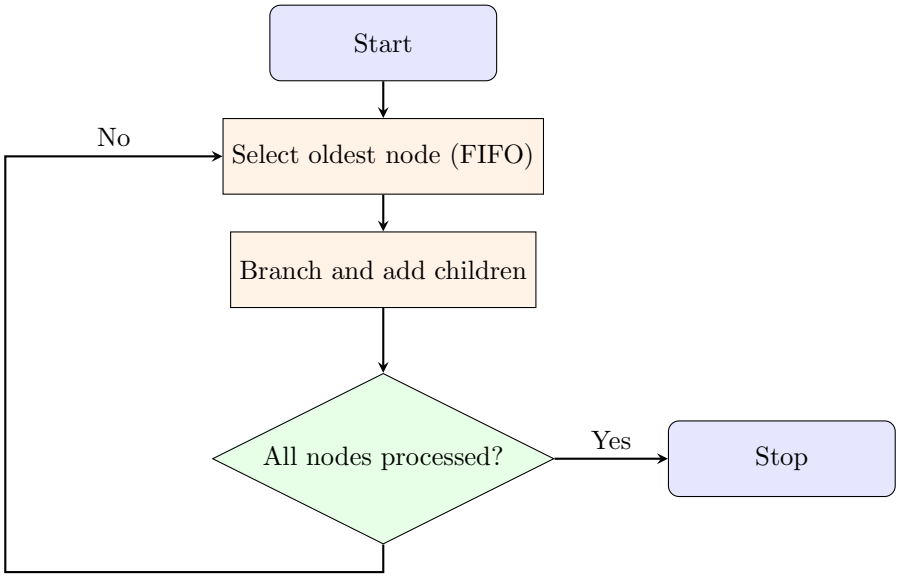

Figure 3: Breadth-First Search (BFS): Selects the oldest node using a First-In, First-Out (FIFO) queue. This explores all nodes at the current depth before going deeper.

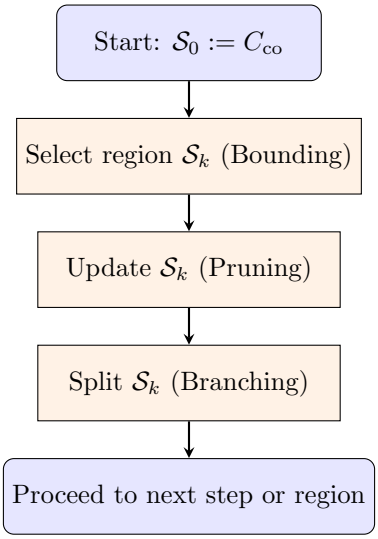

Figure 4: Evolution of the Feasible Region $\mathcal{S}_k$: Starting from continuous relaxation, each region is selected, updated, and possibly split.

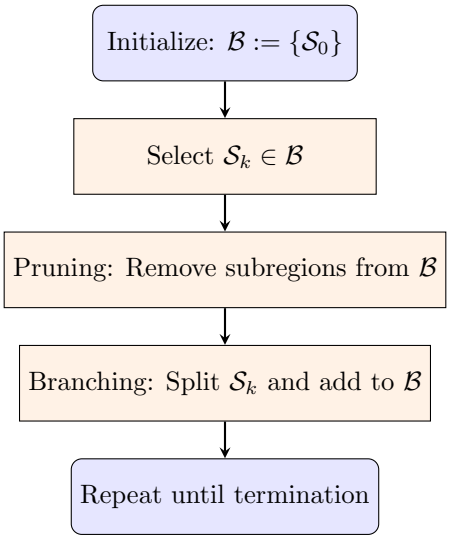

Figure 5: Subregions and Branching Set: $\mathcal{B}$ evolves by pruning unpromising regions and adding new subregions from branching.

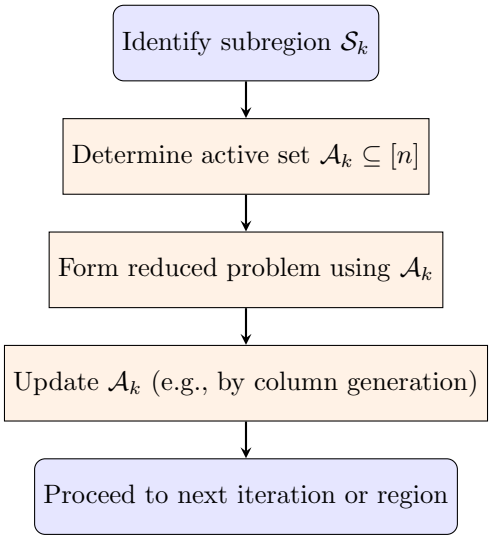

Figure 6: Active set $\mathcal{A}_k$ defines a reduced problem for subregion $\mathcal{S}_k$ and may evolve via column generation, which is discussed in Subsection 2.2, below.

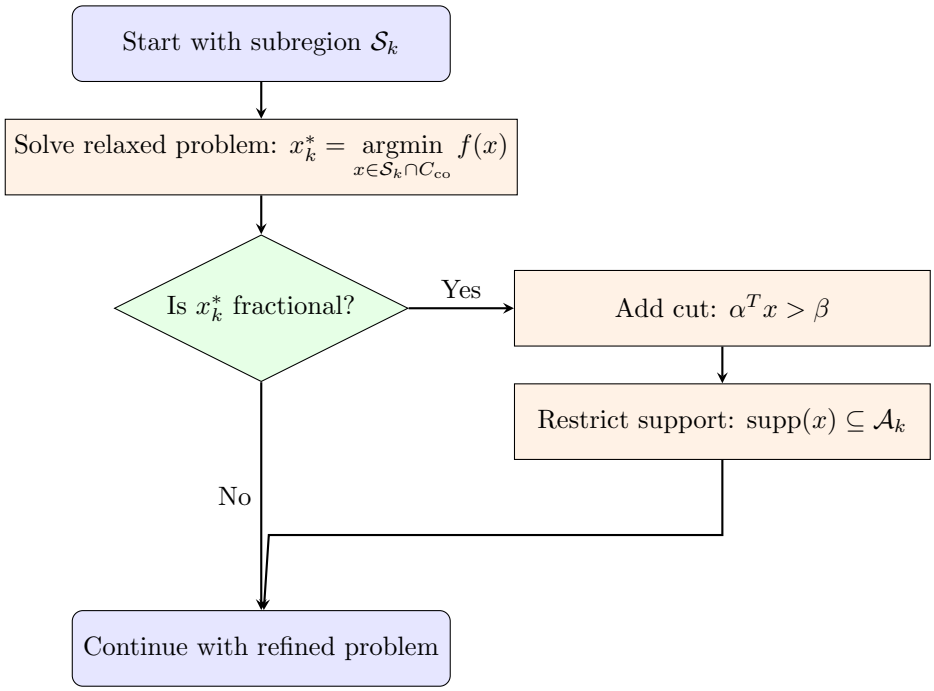

Figure 7: Relaxed Subproblem: Solving the continuous relaxation, refining with a cut if fractional, and restricting with the active set $\mathcal{A}_k$.

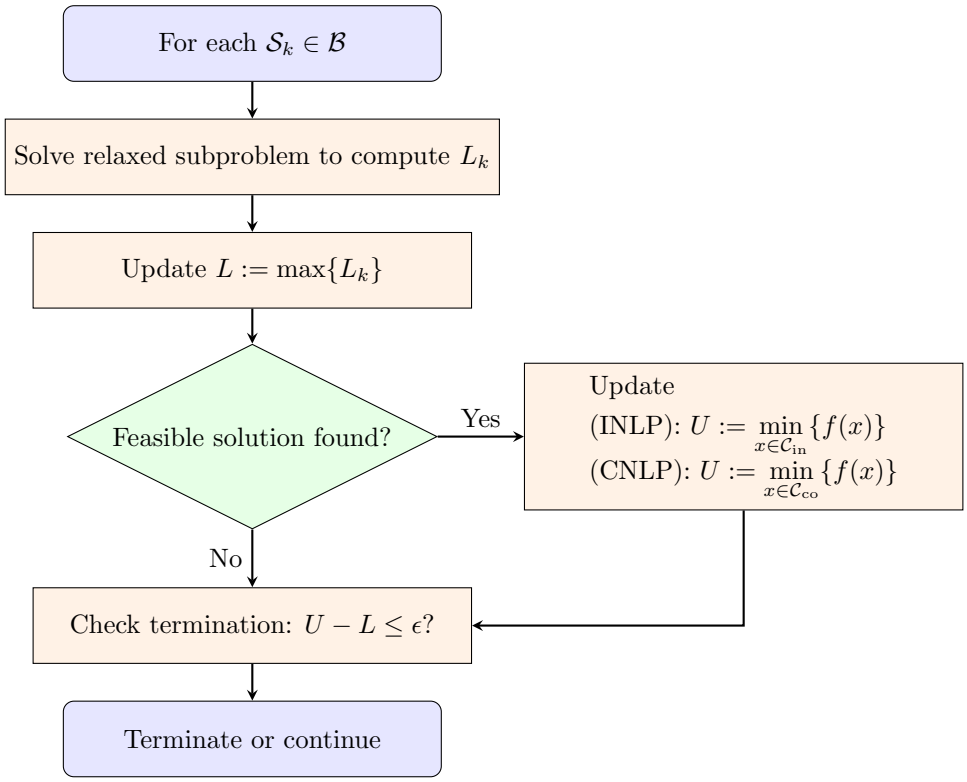

Figure 8: Unified Bound Update Logic for INLP and CNLP: Compute subproblem bounds L_k and update global bounds L and U based on feasible solutions.

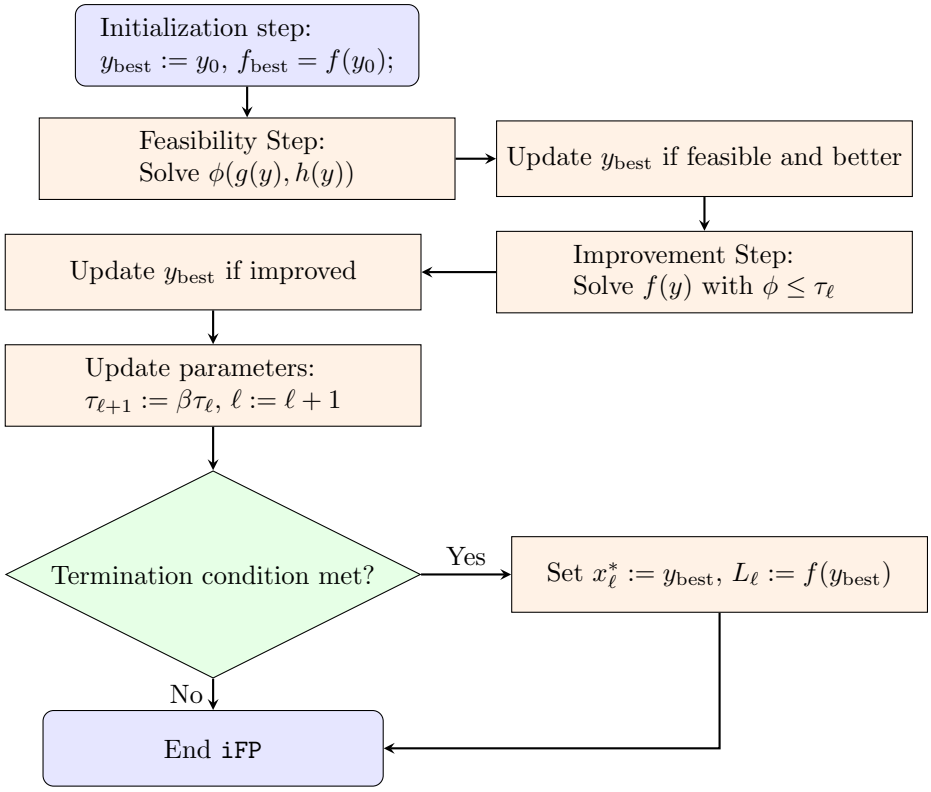

Figure 9: `iFP` Algorithm: Iterative procedure alternating feasibility and improvement steps to find a high-quality integer feasible point. Here $\phi = \phi(g(y), h(y))$ measures the violation of equality and inequality constraints $g(y) = 0$ and $h(y) \leq 0$.

2.2 Expanded Algorithmic Steps and Procedural Details

The three techniques **cutting plane** (`CP`), **column generation** (`CG`), and **feasibility pump** (`FP`) can improve the bounding step of a `BB` algorithm to solve CNLP and INLP problems:

- `CP` refines the relaxation and provides stronger bounds so that the number of branches to be explored is reduced, leading to low computational cost.
- `CG` handles large-scale problems by reducing the space of variables.
- `FP` finds a high-quality solution to a stronger optimization problem.

Several state-of-the-art solvers use these techniques:

- `CPLEX` of IBM [24], `Gurobi` of Gurobi Optimization Group [16], and `SCIP` of SCIP Optimization Suite [15] use various integer and continuous `BB` algorithms together with `CP` and `FP`.
- Both `COIN-OR` [11] and `AMPL` [30] provide strong foundations for implementing `CG`.

We denote an integer `CP` by `iCP` and its continuous version by `cCP`, an integer `CG` by `iCG` and its continuous version by `cCG`, and an integer `FP` by `iFP` and its continuous version by `cFP`.

Here we describe how `iFP` works. Figure 9 shows a flowchart for `iFP`. If the solution x_k^* of the relaxed problem ([26, (15)]) is not an integer, `iFP` is performed in a finite number of iterations to find a high-quality solution of the stronger optimization problem ([26, (17)]). The main steps of `iFP` are **feasibility step**, **improvement step**, and **updating parameters**. `iFP` alternately performs these steps until a high-quality solution of the stronger optimization problem ([26, (17)]) is found:
• **Initialization step** of `iFP`. An initial integer point $y_0 \in C_{\text{in}}$ (e.g., obtained by relaxing integer constraints or randomly rounding a continuous solution) and an initial feasibility threshold $\tau_0 > 0$ are chosen. The iteration counter $\ell := 0$, $y_{\text{best}} := y_0$, and $f_{\text{best}} = f(y_0)$ are initialized.
• **Feasibility step** of `iFP`. The **feasibility subproblem**

$$y_{\ell+1} := \underset{y}{\text{argmin}}\ \phi(g(y), h(y)), \quad \text{s.t.} \quad y_i \in s_i\mathbb{Z} \quad \text{for all} \quad i \in [n]$$

is formed and solved to minimize constraint violations. Here $y_i \in s_i\mathbb{Z}$ imposes integer constraints for each variable and $\phi(g(y), h(y))$ measures the violation of equality and inequality constraints $g(y) = 0$ and $h(y) \leq 0$. If $y_{\ell+1} \in C_{\text{in}}$ and

$f(x_{\ell+1}) < f(y_{\text{best}})$, $y_{\text{best}} := y_{\ell+1}$ is updated.
• **improvement step** of `iFP`. The improvement subproblem

$$\overline{y}_{\ell+1} := \underset{y}{\text{argmin}} \ f(y), \quad \text{s.t.} \quad \phi(g(y), h(y)) \leq \tau_\ell, \quad y_i \in s_i\mathbb{Z} \ \text{ for all } \ i \in [n]$$

is solved. If $\overline{y}_{\ell+1} \in C_{\text{in}}$ and $f(\overline{y}_{\ell+1}) < f(y_{\text{best}})$, $y_{\text{best}} := \overline{y}_{\ell+1}$ is updated. In the update parameters step, the feasibility tolerance $\tau_{\ell+1} := \beta\tau_\ell$ is updated, for some $0 < \beta < 1$ to progressively tighten the tolerance. Then, $\ell := \ell + 1$ is updated. The algorithm is terminated if $y_{\text{best}} \in C_{\text{in}}$ with $\phi(g(y_{\text{best}}), h(y_{\text{best}})) \leq \epsilon$, or no significant improvement in $f(y)$ is observed. If `iFP` yields an improved integer-feasible point, the algorithm updates $x^*_\ell := y_{\text{best}}$ and $L_\ell := f(y_{\text{best}})$.

The same applies to `cFP`, with the distinction that when a relaxed problem is infeasible, `cFP` can still be employed to discover an improved feasible solution.

Recent work further explores differentiable and learning-assisted feasibility pumps, e.g., Cacciola et al. [5], which integrate gradient-based learning into classical `FP` heuristics.

2.3 Software for MINLP

This section discusses five state-of-the-art solvers for the MINLP problem (7): `BARON` [46], `ANTIGONE` [34], `LINDO` [6], `Couenne` [1], and `KNITRO` [4]. Among them, `BARON`, `ANTIGONE`, `LINDO`, and `Couenne` are global solvers that combine local and global methods, while `KNITRO` is a local solver. Except for `KNITRO`, which operates as a stand-alone CNLP/MINLP solver embedding its own local optimization routines, the other solvers rely on external engines such as `IPOPT` of Wächter and Biegler [43] or `SNOPT` of Gill et al. [14] to solve NLP subproblems, and `CPLEX` [24], `Gurobi` [16], `Xpress` [9], or `GLPK` [32] to solve LP and MILP subproblems. Recently, Zhang & Sahinidis [46] provided an extensive numerical comparison among many state-of-the-art optimization solvers (including the above-mentioned solvers) for solving CNLP, INLP, and MINLP problems, and proposed a learning-based framework using graph convolutional networks to dynamically deactivate probing in `BARON`, yielding significant reductions in solution time across benchmark MINLP libraries.

CNLP solvers. In the case of CNLP, where no integer variables are involved, the goal is to efficiently solve nonlinear programs subject to general nonlinear constraints. Solvers such as `IPOPT`, `SNOPT`, `KNITRO`, and `CONOPT` are widely used for this purpose, either as stand-alone solvers or as subroutines within larger MINLP frameworks.

`IPOPT` implements an interior-point method that combines a line search strategy with a filter technique to ensure globalization and robust convergence. `SNOPT` is based on sequential quadratic programming (SQP), which solves a sequence

of quadratic programming subproblems. It uses a reduced-Hessian active-set method to exploit problem structure and sparsity, a line search for convergence, and a limited-memory quasi-Newton approach to approximate the Hessian of the Lagrangian.

For solving LP and MIP, solvers such as `CPLEX`, `Gurobi`, `Xpress`, and `GLPK` are often used as components within both CNLP and MINLP solvers. `CPLEX`, `Gurobi`, and `Xpress` support various methods for LPs, including primal simplex, dual simplex, and barrier (interior-point) methods, as well as advanced `BB` algorithms for MIPs. Although they differ in implementation details and performance tuning, they are capable of solving large-scale LP and MIP instances efficiently. In contrast, `GLPK` relies on a primal simplex method for LPs and a basic `BB` approach for MIPs, making it more suitable for small- to medium-scale problems.

These solvers are critical not only for the direct solution of CNLPs and LP/MIP problems but also as building blocks for hybrid solvers that address more complex MINLP problems.

A Generic MINLP Solver Workflow. Algorithm 1 outlines a high-level software framework for solving the MINLP problem (7). It describes typical preprocessing, structure detection, and optimization phases used inside modern MINLP solvers. It has three steps: reformulation of input, detection of special structure, and applying local or global optimization algorithms to alternately find the local or global minimizer of the MINLP problems.

Algorithm 1 A Generic Solver Workflow for the MINLP Problem (7)

Reformulation of input: ($S0_1$) Convert the input problem into a standardized or enhanced form.

Detection of special structure: ($S1_1$) Find aspects of the problem (e.g., convexity, sparsity, and separability) to use for specific optimization methods.

repeat

Applying optimization algorithms: ($S2_1$) Call local or global optimization algorithms to solve the MINLP problem (7).

until the stopping criterion is met

`BARON`. This solver performs the steps ($S0_1$) and ($S1_1$) of Algorithm 1, and then alternately executes step ($S2_1$) until a global minimizer is found. In ($S0_1$), all integer variables are relaxed to continuous bounds to obtain an initial relaxation; an automatic algebraic reformulation is carried out to eliminate redundancies, identify equivalences, and simplify the model. All nonconvex terms are replaced by tight convex relaxations derived from factorable reformulations, and interval arithmetic is employed to refine variable bounds and verify feasibility. In

($S1_1$), `BARON` analyzes the convexity or nonconvexity of functions, detects decomposability, classifies constraints by type, and identifies dependencies among variables. In ($S2_1$), a branch-and-bound algorithm is applied, augmented with domain-reduction techniques, convex relaxations for lower bounds, interval analysis for bound tightening, and heuristics and cutting planes to improve upper bounds and feasibility. `BARON` interfaces with external solvers such as `CPLEX`, `Gurobi`, and `Xpress` to solve LP and MILP relaxations, and `IPOPT`, `CONOPT`, or `KNITRO` to solve NLP subproblems [46]. It provides deterministic guarantees of global optimality and has demonstrated state-of-the-art performance across standard MINLP benchmark libraries. Recent `BARON` versions (24–25) include significant improvements in presolve, convexity detection, and robust handling of rounding errors.

Recent developments by Zhang & Sahinidis [47] extended `BARON` by integrating a machine-learning-based module that uses graph convolutional networks (GCNs) to adaptively control the use of probing—a key domain-reduction technique. The GCN represents each MINLP instance as a tripartite graph of variables, constraints, and nonlinear operators, learning when to deactivate probing to balance computational cost and tightening efficiency. This learning-guided probing policy achieves significant average speedups (approximately 40% for MINLPs and 25% for NLPs) while preserving `BARON`'s robustness and deterministic global optimization guarantees.

`ANTIGONE`. This solver performs the steps ($S0_1$) and ($S1_1$) of Algorithm 1, and then alternately executes step ($S2_1$) until a global minimizer is found. In ($S0_1$), an automatic reformulation decomposes the problem into simpler forms, removes redundant constraints and variables, and reformulates the objective function and constraints to expose convex, linear, or separable structures. All nonconvex functions are replaced by convex underestimators and concave overestimators, including McCormick and outer-approximation relaxations for bilinear, quadratic, trilinear, and signomial terms. The feasible region is reduced by bound-tightening heuristics, and initial dual bounds based on convex relaxations are computed to guide the optimization process. In ($S1_1$), `ANTIGONE` determines whether each function is convex, concave, or nonconvex; detects decomposability for independent subproblems; replaces nonlinear expressions with suitable relaxations or piecewise-linear approximations; classifies constraints for tailored treatment; identifies variable dependencies for branching; and adds custom cutting planes to strengthen relaxations. In ($S2_1$), a branch-and-bound algorithm enhanced with cutting planes and validated interval arithmetic is applied to ensure global optimality. `ANTIGONE` interfaces `CPLEX` for LP and MIP relaxations, `CONOPT` or `SNOPT` for local NLP subproblems, and `Boost` for validated interval arithmetic [34].

`KNITRO`. This solver performs the steps ($S0_1$) and ($S1_1$) of Algorithm 1, and iteratively applies local optimization algorithms in ($S2_1$) to obtain a local minimizer of the MINLP problem. In ($S0_1$), all variables and constraints are automatically scaled to improve numerical stability, the objective and constraint functions

are normalized to ensure consistent gradient behavior, and variable bounds are tightened to enhance computational efficiency. In ($S1_1$), the solver identifies convex or nonconvex components, exploits sparsity in the Jacobian and Hessian matrices, and uses efficient sparse data structures to reduce memory and time requirements for large-scale problems. It also detects integer and binary variables, dynamically integrating them within a branch-and-bound framework that combines continuous NLP optimization with discrete search. In ($S2_1$), an interior-point (barrier) algorithm is used by default to solve the continuous relaxation, while active-set and trust-region methods are available as alternatives. The active-set method identifies active constraints and solves a sequence of quadratic programming subproblems, whereas the trust-region method constructs quadratic models whose solutions are restricted to controlled step sizes to ensure stability. `KNITRO` solves MINLPs using a local branch-and-bound scheme combined with its built-in continuous optimization methods, offering efficient local CNLP/MINLP solutions without global optimality guarantees [4].

`LINDO`. This solver also performs the steps ($S0_1$) and ($S1_1$) of Algorithm 1 and then alternately executes step ($S2_1$) until a global minimizer is found. In ($S0_1$), as with the previously described solvers, variables and constraints are automatically scaled to improve numerical conditioning, and presolve procedures aggregate redundant constraints and simplify the model to reduce its size. Integer and nonlinear constraints may be relaxed, and nonlinear or nonsmooth functions are automatically replaced by mathematically equivalent linear approximations through LINDO's built-in linearization tools. In ($S1_1$), the problem structure is analyzed to determine convexity, nonconvexity, linearity, and sparsity, enabling the selection of the most efficient algorithm. In ($S2_1$), interior-point or simplex methods are used to solve LP subproblems, while SQP or interior-point algorithms are applied to NLPs. For MINLPs, a branch-and-bound algorithm with convex and interval relaxations is employed, combining continuous optimization (SQP or barrier methods) with discrete search. Modern versions (e.g., `LINGO` 21 and later) also include built-in global, multistart, and stochastic solvers, allowing full MINLP solution without external engines [6]. Recent `LINDO` versions include improved MIP symmetry detection and stronger global solver enhancements.

`Couenne`. This solver also performs the steps ($S0_1$) and ($S1_1$) of Algorithm 1, and then alternately executes step ($S2_1$) until a global minimizer is found. In ($S0_1$), all nonconvex expressions are replaced by convex underestimators or piecewise-linear relaxations, such as McCormick convex envelopes, and the variables and constraints are scaled to improve numerical stability. In ($S1_1$), structural properties of the problem are exploited to increase efficiency: Couenne identifies convex, concave, and nonconvex components, exploits sparsity in constraint and Jacobian matrices, classifies variables and constraints, and decomposes complex formulations into simpler subproblems. In ($S2_1$), a spatial branch-and-bound algorithm is applied, enhanced with convex relaxations, cutting planes, range tightening, and domain-reduction heuristics to ensure global optimality.

Throughout these steps, `Couenne` employs two key techniques: *range reduction* and *constraint propagation*. Range reduction tightens variable bounds and reduces the search space in ($S0_1$), facilitates convexity and separability detection in ($S1_1$), and refines bounds dynamically at each node in ($S2_1$). Constraint propagation simplifies variable domains using the model constraints in ($S0_1$), exposes structural relationships in ($S1_1$), and iteratively prunes infeasible branches and strengthens relaxations in ($S2_1$). `Couenne` interfaces `IPOPT` for NLP relaxations and `CPLEX`, `Gurobi`, or `GLPK` for LP and MILP subproblems [1].

Starting from version 11, `Gurobi` includes a deterministic spatial branch-and-bound framework with convex relaxations and automatic bound tightening for solving MINLPs, providing global optimization capabilities. While its MINLP engine is based on classical rule-based methods rather than ML-guided branching or cut selection, it should be regarded as a modern global MINLP solver; see [17, 18].

2.4 Theoretical Guarantees of Global Convergence

For the MINLP problem (7) defined over the mixed-integer nonlinear feasible set $C_{\text{mi}} \subseteq \mathbf{X}$, classical `BB` methods and their modern enhancements (cutting planes, column generation, and feasibility pumps) are theoretically guaranteed to converge to an ϵ-global optimum under standard assumptions [21, Ch. 3], [10, Ch. 4], [20, Ch. 2–3]. In particular, if the feasible set is compact, the branching strategy exhaustively partitions the domain, and the bounding mechanism is valid, the `BB` algorithm will terminate with an ϵ-global optimal solution, defined below.

Definition 2.1 (ϵ-Global Optimal Solution)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function of the MINLP problem (7) with feasible set $C_{\text{mi}} \subseteq \mathbf{X}$. A feasible point $x^* \in C_{\text{mi}}$ is called an* ϵ-global optimal solution *if its objective function value is within $\epsilon \geq 0$ of the global minimum, i.e.,*

$$f(x^*) \leq \inf_{x \in C_{\text{mi}}} f(x) + \epsilon.$$

If $\epsilon = 0$, the solution x^* is a global optimal solution*.*

Remark 2.1. *The above definition naturally applies to all special cases of MINLP:*

- ***INLP case:*** *If the index set of continuous variables is empty ($J = \emptyset$), then C_{mi} reduces to the integer nonlinear feasible set C_{in} in (3). An ϵ-global optimal solution $x^* \in C_{\text{in}}$ satisfies*

$$f(x^*) \leq \inf_{x \in C_{\text{in}}} f(x) + \epsilon.$$

- ***CNLP case:*** *If the index set of integer variables is empty ($I = \emptyset$), then C_{mi} reduces to the continuous nonlinear feasible set C_{co} in (9). An ϵ-global optimal solution $x^* \in C_{\text{co}}$ satisfies*

$$f(x^*) \leq \inf_{x \in C_{\text{co}}} f(x) + \epsilon.$$

Thus, the ϵ-global optimality concept unifies all continuous, integer, and mixed-integer nonlinear optimization settings under a single definition.

Lemma 2.1 (Global Optimality Gap)**.** *Let $\{L_k\}$ and $\{U_k\}$ be the lower and upper bounds computed by a* `BB` *algorithm for the MINLP problem (7), and define*

$$U^* := \min_k U_k, \quad L^* := \max_k L_k.$$

Then the quantity

$$G := U^* - L^*$$

is called the **global optimality gap***. It satisfies*

$$0 \leq f(x^*) - f^* \leq G,$$

where x^* is the best feasible solution found (attaining U^*) and $f^* = \inf_{x \in C_{\text{mi}}} f(x)$ is the global minimum.

Proof. By the definition of U^*, there exists a feasible $x^* \in C_{\text{mi}}$ such that $f(x^*) = U^*$. By the validity of the lower bounds, $L_k \leq \inf_{x \in \mathbf{X}_k \cap C_{\text{mi}}} f(x)$ for all k, which implies $L^* \leq f^*$. Hence

$$f(x^*) - f^* = U^* - f^* \leq U^* - L^* = G.$$

Nonnegativity $f(x^*) \geq f^*$ implies $f(x^*) - f^* \geq 0$. □

Theorem 2.1 (Global Convergence of Enhanced `BB` Algorithms for MINLP)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuous and consider the MINLP problem*

$$\min_{x \in C_{\text{mi}}} f(x)$$

with feasible set

$$C_{\text{mi}} := \{x \in \mathbf{X} \mid g(x) = 0, \quad h(x) \leq 0, \quad x_i \in s_i\mathbb{Z} \text{ for } i \in I,\, x_i \in \mathbb{R} \text{ for } i \in J\},$$

where $\mathbf{X}$ is the simple box defined in (1), and $I \cup J = [n]$. Suppose an enhanced `BB` *algorithm (branch-and-cut/price) is applied. Assume:*

(i) ***Bounded Feasibility:*** *C_{mi} is nonempty and compact; f is bounded below on C_{mi}.*

(ii) ***Exhaustive Partitioning:*** *The branching process generates disjoint subregions* $\{\mathbf{X}_k\}$ *such that*

$$\bigcup_k \mathbf{X}_k = \mathbf{X}, \quad \lim_{k\to\infty} \operatorname{diam}(\mathbf{X}_k) = 0.$$

(iii) ***Valid Bounding:*** *For each* $\mathbf{X}_k$*, the algorithm computes*
- *a lower bound* $L_k \le \inf_{x\in\mathbf{X}_k\cap C_{\mathrm{mi}}} f(x)$,
- *an upper bound* $U_k = f(x_k)$ *for some feasible* $x_k \in \mathbf{X}_k \cap C_{\mathrm{mi}}$.

(iv) ***Cutting-Plane Validity:*** *All cuts are valid and do not remove any globally feasible solution.*

(v) ***Column-Generation Correctness:*** *Restricted master problems are solved to optimality with valid dual bounds.*

(vi) ***Termination Criterion:*** *The algorithm terminates when the global optimality gap*

$$G = U^* - L^* < \epsilon$$

for a prescribed $\epsilon > 0$.

Then the `BB` *algorithm terminates in finitely many steps with an* ϵ*-global optimal solution* $x^* \in C_{\mathrm{mi}}$ *satisfying*

$$f(x^*) \le \inf_{x\in C_{\mathrm{mi}}} f(x) + \epsilon.$$

Proof. By (i)–(ii), the feasible set is compact and the branch diameters tend to zero. By (iii)–(v), the sequences $\{L^*\}$ and $\{U^*\}$ are valid and monotone, and no feasible solution is ever discarded. By Lemma 2.1, the global optimality gap satisfies

$$0 \le f(x^*) - f^* \le U^* - L^* = G.$$

Once $G < \epsilon$, i.e., (vi) holds, we have $f(x^*) \le f^* + \epsilon$, so x^* is an ϵ-global optimal solution. Finite termination follows from standard `BB` theory [21, Thm. 3.3.1] and [10, Ch. 4, 11]. □

Analytical remark. The convergence results summarized above restate the classical theory of branch-and-bound [10, 21]. In this work, they are explicitly extended to encompass data-driven bounding and branching decisions, thereby connecting deterministic convergence theory with learning-augmented optimization. This unified interpretation provides the analytical synthesis linking classical heuristics and learned policies under the same $(\mathcal{S}, \mathcal{A}, \pi_\theta)$ abstraction.

2.5 Performance of Commercial Solvers on Standard MINLPs

Table 1 presents the performance of several commercial MINLP solvers on standard benchmark problems, including crew scheduling, knapsack, vehicle routing, facility location, energy grid optimization, and hydropower scheduling.

These problems are categorized by problem size, where **small** refers to instances with fewer than 100 variables, **medium** refers to 100–1000 variables, and **large** refers to more than 1000 variables. The table highlights the compatibility of problems with general-purpose MINLP solvers such as `BARON`, `ANTIGONE`, `LINDO`, `Couenne`, and `KNITRO`; recent advances also include machine-learning-enhanced frameworks (e.g., the GCN-based probing control in `BARON`) and the deterministic global MINLP capability in `Gurobi` version 11 and later.

While these solvers can handle integer variables and nonlinear constraints, they are not specifically tailored to guarantee exact solutions for purely discrete problems. Instead, they are often used when modelling flexibility or integration of nonlinear features is required.

These benchmark categories are widely used in comparative MINLP studies such as MINLPLib 2 and QPLIB, allowing consistent assessment of solver scalability and robustness.

Among these solvers, `BARON` and `ANTIGONE` typically achieve the most reliable global results on medium- and large-scale problems, while `Couenne` provides a strong open-source alternative for research and smaller instances. `LINDO` and `KNITRO` are well suited for local or convex MINLPs and provide efficient convergence for large smooth problems.

2.6 Recommendation and Conclusion

Table 2 provides a comparative overview of several well-known solvers for MINLP problems, with a focus on their scalability, exactness, and support for high-performance computing (HPC). The entries indicate qualitative assessments over problem sizes (small, medium, large), whether the solver guarantees global optimality ("Exact?"), and whether parallel or distributed computing is supported ("HPC?"). Solvers such as `BARON` and `ANTIGONE` are designed for global optimization and can certify optimality for nonconvex MINLPs, thus marked as exact. `KNITRO`, in contrast, is a local solver and does not provide global optimality guarantees, though it remains effective for local nonlinear problems. `Couenne` provides deterministic global optimization for nonconvex MINLPs via convex relaxations but is exact only within convex subproblems. The classification reflects practical trade-offs between solver robustness, computational cost, and algorithmic guarantees, helping to guide solver selection based on application-specific needs.

problem	size			software
	small	medium	large	
CSP (Crew Scheduling)	+	+	+	ANTIGONE, BARON, KNITRO
KP (Knapsack Problem)	+	+	+	BARON, Couenne, KNITRO
VRP (Vehicle Routing)	+	+	±	BARON, ANTIGONE
FLP (Facility Location)	+	+	±	BARON, ANTIGONE, Couenne
EGO (Energy Grid Optimization)	+	+	±	BARON, ANTIGONE, LINDO, Gurobi
HS (Hydropower Scheduling)	+	+	±	ANTIGONE, BARON, LINDO, KNITRO, Gurobi

Table 1: Performance of selected commercial MINLP solvers on standard benchmark problems. The size categories are defined as small: 1–100 variables, medium: 100–1000 variables, and large: > 1000 variables. While these solvers handle integer variables and nonlinear constraints, they do not guarantee exact solutions for purely discrete problems. Recent advances, however, have introduced learning-guided domain-reduction strategies (e.g., BARON 11) and deterministic global MINLP extensions in Gurobi 11+.

solver	problem size			exact?	HPC?	software/libraries/references
	small	medium	large			
BARON	+	+	±	+	±	Global nonconvex MINLP solver with ML-guided probing https://minlp.com/baron [46, 47]
ANTIGONE	+	+	±	+	±	Global optimization for MINLP https://antigone.aimms.com [34]
KNITRO	+	+	±	−	±	Local NLP/MINLP solver https://www.artelys.com/knitro/ [4]
LINDO	+	+	±	+	±	Global MINLP solver with parallel support https://www.lindo.com [6]
Couenne	+	+	±	+*	±	Open-source global MINLP solver (spatial BB with convex relaxations) https://coin-or.github.io/Couenne/ [1]

Table 2: Classification of MINLP solvers. "Exact?" refers to the ability to certify global optimality. Couenne is exact only within convex subproblems (*), providing deterministic global optimization via convex relaxations. "HPC?" indicates support for parallel or distributed computing.

The classical exact methods presented in this section form the algorithmic backbone upon which learning-based enhancements operate. In Sections 4.1–4.2, we revisit these steps—bounding, branching, node selection, and pruning—through the lens of data-driven decision-making, where machine learning models learn to approximate or refine these operations while preserving the exactness guarantees established here.

Recent examples include the learning-guided probing policy in `BARON` [47], demonstrating how data-driven components can accelerate exact global frameworks.

3 Previous Surveys on ML for Exact MINLP Methods

Recent research at the intersection of ML and mathematical optimization has led to the development of learning-augmented solvers for ILP, MILP, CNLP, and MINLP. These methods improve solver efficiency by replacing or enhancing traditional decision heuristics, such as branching, cut selection, and node ordering, with data-driven models. RL has been particularly effective for learning sequential decision-making policies within `BB` frameworks, while supervised learning (SL) and imitation learning (IL) have been employed to guide primal heuristics, feasibility estimation, or constraint generation. In CNLP, policy optimization methods from continuous control have been adapted to preserve feasibility and ensure global convergence. This section surveys recent advances in ML and RL for exact optimization, with a focus on methods that are integrated into solver pipelines and that preserve the correctness, convergence, or optimality guarantees of classical algorithms (see Figure 10 and related discussions, below).

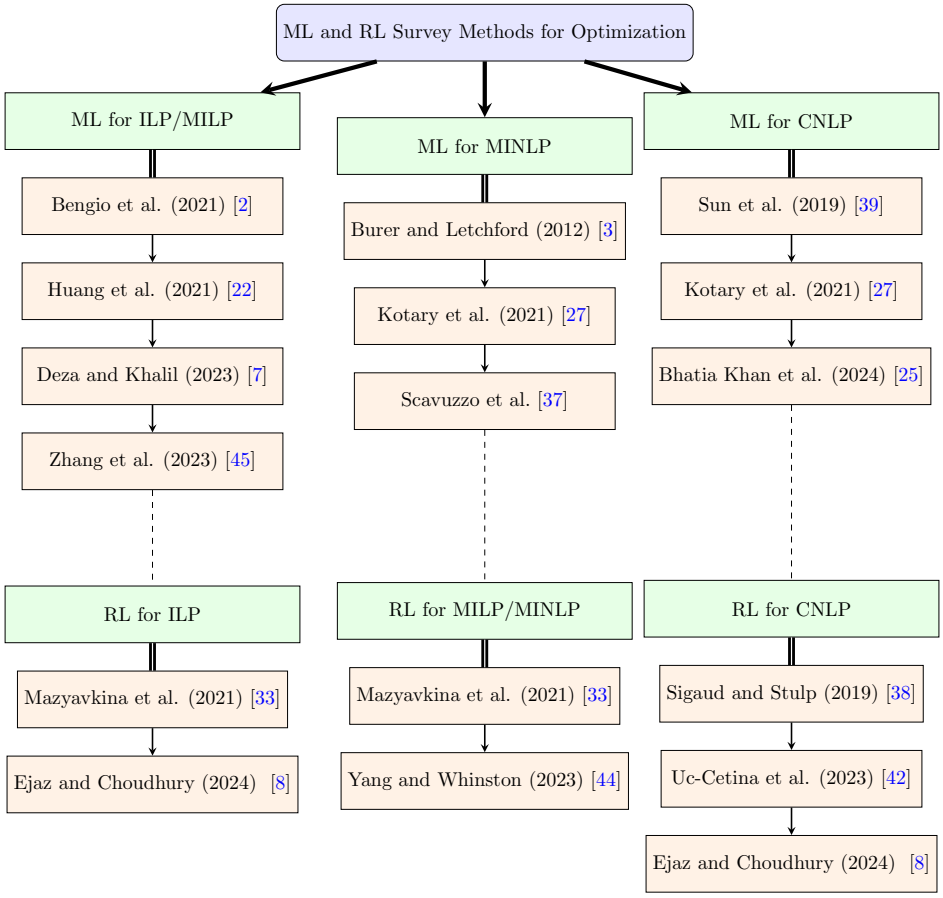

Figure 10: Flowchart Summary of Survey Papers on ML and RL Methods for ILP, MILP, MINLP, and CNLP Problems.

ML Methods for ILP and MILP Problems. Deza and Khalil [7] present a comprehensive survey on ML-guided cut selection in MILPs, categorizing prediction tasks (e.g., ranking, classification) and outlining unique challenges in generalizing these methods to nonlinear relaxations in MINLPs.

Bengio et al. [2] offer a broad methodological tour d'horizon of ML for combinatorial optimization, covering SL, IL, and RL techniques for ILP/MILP solvers. They emphasize the role of graph neural networks (GNNs), policy learning strategies, and training pipelines, and identify core open issues regarding generalization, data efficiency, and the lack of theoretical assurances.

Mazyavkina et al. [33] survey RL-based strategies for exact combinatorial optimization, detailing RL policies for branching, cut selection, and node order in ILP/MILP. They highlight empirical solver acceleration, transferability of learned policies, and adaptability to instance distributions. However, the survey also underscores limitations related to data dependence, unstable training dynamics, and limited integration into certified solvers.

Zhang et al. [45] survey the broader landscape of ML for MIP, covering both exact and approximate learning-based strategies across solver components such as branching, cut selection, primal heuristics, and node evaluation. Their work synthesizes recent architectures, including Transformers and GNNs, and emphasizes performance metrics, benchmarking practices, and practical challenges such as solver interpretability and computational overhead.

Huang et al. [22] provide an in-depth review focused specifically on ML-enhanced `BB` techniques for MILPs. They categorize ML approaches by target solver decision points (e.g., variable selection, node selection, pruning, and cut generation), survey neural models used to learn branching scores, and highlight the algorithmic implications of learned branching policies. The survey also identifies persistent research gaps in generalization to unseen distributions, data collection for rare-event learning, and maintaining solver completeness with ML-guided decisions.

ML Methods for MINLP Problems. Burer and Letchford's survey [3] provides a foundational review of algorithms and software for solving nonconvex MINLPs. It categorizes techniques such as `BB`, branch-and-cut, outer approximation, and hybrid decomposition strategies. Although this work predates most ML integration, it establishes the theoretical framework within which ML-enhanced MINLP solvers operate today.

Kotary et al. [27] review end-to-end learning approaches for constrained optimization, including those applicable to MINLPs. Their survey discusses model-based approximations, learned constraint satisfaction, and optimization layers trained via supervised or reinforcement signals. These techniques enable approximate yet feasible solutions in cases where MINLP constraints are only partially known or learned from data. Their review also highlights open questions on generalization, convergence, and robustness of learned surrogates when

embedded into deterministic solvers.

Scavuzzo et al. [37] provide a recent and comprehensive survey on ML-augmented BB frameworks for solving MILPs, with insights relevant to MINLP extensions. Their survey systematically categorizes learning methods applied to core solver components, including branching, cut selection, primal heuristics, node selection, and solver configuration. Particular emphasis is placed on graph-based representations of MILPs, especially bipartite graph encodings and GNNs, as well as strategies for SL and RL to enhance solver performance. The authors highlight critical integration challenges such as balancing solver interpretability with ML-driven efficiency, preserving exactness and convergence guarantees, and ensuring robust generalization across heterogeneous problem instances. The survey concludes with open research directions for extending ML-augmented BB methods to nonlinear and mixed-integer nonlinear problems, bridging classical deterministic solvers with modern data-driven approaches.

ML Methods for CNLP Problems. Sun et al. [39] provide a broad survey of optimization algorithms from a ML perspective, covering first-order, second-order, and derivative-free methods. Although not explicitly framed as a study of CNLP, many of the optimization techniques reviewed—such as stochastic gradient descent, quasi-Newton methods, and derivative-free search—directly apply to CNLP solvers and share similar convergence and landscape-analysis concerns.

Kotary et al. [27] also discuss learning-augmented techniques for CNLP problems. Their survey includes surrogate modelling, differentiable optimization layers, and hybrid policy search strategies used to approximate exact CNLP solvers. These methods preserve feasibility guarantees through learned projections or constraint regularizers, contributing to data-driven CNLP solvers that retain global convergence properties under certain conditions.

Bhatia Khan et al. [25] provide a comprehensive survey of global and local nonlinear optimization methods, including deterministic algorithms such as BB, and stochastic strategies like evolutionary techniques and metaheuristics. While primarily focused on classical CNLP, the review also highlights emerging trends that incorporate ML, such as surrogate modelling, neuro-symbolic optimization, and hybrid policy search. The paper contextualizes ML's role in CNLP as one of improving convergence behaviour, landscape navigation, and global search effectiveness, particularly in high-dimensional or black-box settings. It serves as a valuable bridge between traditional optimization theory and modern data-driven CNLP strategies.

RL Methods for ILP and MILP Problems. Mazyavkina et al. [33] present one of the earliest comprehensive surveys on the application of RL to exact combinatorial optimization, focusing on ILP and MILP. They categorize RL methods by the solver decision components these methods target—such as branching, cut selection, and node selection—and discuss how learned policies have

outperformed traditional, human-engineered heuristics. The advantages they identify include solver speedups, adaptability, and the ability to learn from experience. However, they also highlight key challenges: the need for extensive solver-generated training data, instability in RL training, and limited theoretical guarantees when RL affects core decisions in exact algorithms.

RL Methods for CNLP Problems. Sigaud and Stulp [38] provide a structured survey of policy search algorithms tailored to continuous-action domains. They contrast several approaches—including policy gradient methods, evolutionary strategies, and Bayesian optimization—evaluating each on criteria such as sample efficiency, convergence behaviour, and learning stability. Although these methods were primarily developed for control tasks, they determine that many could function effectively as exact or near-exact solvers for continuous nonlinear programming problems, offering principled alternatives to classical optimization techniques.

Uc-Cetina et al. [42] present a comprehensive survey of reinforcement learning (RL) methods applied to *natural language processing* (hereafter, **NLProc**) tasks such as dialogue management, machine translation, and text generation. While the survey is not concerned with mathematical programming or continuous optimization per se, the RL algorithms it reviews—including policy-gradient, actor–critic, and reward-shaping techniques—are fundamentally instances of CNLP applied to expected return functions. This methodological overlap provides a useful conceptual bridge between RL advances in NLProc and CNLP: the same gradient-based policy optimization strategies that tune linguistic models are also central to data-driven approaches for nonlinear and mixed-integer nonlinear optimization. Hence, although domain-specific, the work of Uc-Cetina et al. highlights optimization paradigms that are increasingly shared across machine learning and mathematical programming research.

RL Methods for MILP and MINLP Problems. Mazyavkina et al. [33] present a comprehensive survey on RL applications to exact combinatorial optimization, focusing on ILP and MILP problems. They categorize RL methods by solver decision components (e.g., branching, cut selection, node selection, primal heuristics) and discuss how learned policies have outperformed human-engineered heuristics. The survey highlights benefits like solver speedups, adaptability, and experience-driven learning, but also emphasizes challenges such as the requirement for extensive solver-generated training data, instability in RL training, and limited theoretical guarantees when core solver components are learned.

Yang and Whinston [44] provide a detailed survey of RL methods for classical combinatorial optimization problems such as the traveling salesperson and quadratic assignment problems. While their focus is on discrete optimization rather than solver integration, the RL frameworks they analyze—including Q-learning, actor–critic, and deep RL architectures—are directly transferable to decision processes within MILP and MINLP formulations.

RL Methods for LP, ILP, MILP Problems. Ejaz and Choudhury [8] review the use of RL and ML for resource allocation and scheduling problems in 5G and beyond networks, many of which are formulated as LP, ILP, or MILP models. Their survey emphasizes domain-specific adaptations of RL to guide heuristic and approximate optimization, offering insight into how learning-based strategies can complement exact solver logic in large-scale, dynamic environments.

A structured overview of key survey papers in this area is provided in Table 3, which organizes the literature by focus area and problem type. Together, these surveys offer a comprehensive foundation for understanding how ML and RL techniques are being integrated into exact optimization pipelines. They span a diverse set of solver components and problem classes, and collectively highlight both the promise of learning-augmented methods and the challenges of preserving correctness, generalization, and scalability in real-world settings.

Table 3: Summary of Surveyed ML and RL Methods for Exact Optimization

Survey	**Focus Area**	**Problem Type**
Deza & Khalil (2023) [7]	ML-guided cut selection	MILP
Bengio et al. (2021) [2]	ML for combinatorial optimization	ILP, MILP
Zhang et al. (2023) [45]	ML for MIP solver components	MIP
Huang et al. (2021) [22]	ML-enhanced BB for MILP	MILP
Burer & Letchford (2012) [3]	Classical nonconvex MINLP solvers	MINLP
Kotary et al. (2021) [27]	End-to-end learning for constrained optimization	MINLP, CNLP
Scavuzzo et al. (2024) [37]	ML-augmented BB frameworks	MILP, MINLP
Sun et al. (2019) [39]	Optimization methods in ML	CNLP
Bhatia Khan et al. (2024) [25]	Classical and ML-enhanced CNLP methods	CNLP
Sigaud & Stulp (2019) [38]	Policy search algorithms for continuous control	CNLP
Mazyavkina et al. (2021) [33]	RL strategies for exact combinatorial optimization	ILP, MILP
Uc-Cetina et al. (2023) [42]	RL methods for *natural language processing* (NLProc) tasks	Methodological link to CNLP
Yang & Whinston (2023) [44]	RL for combinatorial optimization	MILP, MINLP
Ejaz & Choudhury (2024) [8]	RL for LP/ILP/MILP in networks	ILP, MILP

Positioning and Contribution Beyond Prior Surveys. While the surveys summarized above provide detailed coverage of learning-based enhancements for specific solver components or optimization paradigms, they generally remain fragmented—addressing either discrete (ILP/MILP) or continuous (CNLP) domains, or focusing on isolated learning frameworks such as supervised or reinforcement learning. In contrast, the present work unifies these perspectives by (i) explicitly connecting classical exact global-optimization algorithms (discussed in Section 3 of the main paper [26]) with data-driven augmentation strategies applied across solver components; (ii) synthesizing ML and RL approaches

within a common branch-and-bound framework that preserves theoretical guarantees of convergence and global optimality; and (iii) situating these advances within the landscape of contemporary industrial solvers. Sections 4–5 of the main paper [26], where their algorithms are discussed here in Section 4, therefore move beyond prior reviews to develop an integrative taxonomy of learning-augmented decision mechanisms, a comparative analysis of neural architectures for branching, cutting, and parameter control, and a forward-looking discussion of current limitations, emerging trends, and pathways toward production-level adoption in MINLP solver ecosystems.

4 Neural Networks (NNs) as Branch-and-Bound Enhancers

This section provides background on both ML and RL, along with pseudocode for all ML and RL methods discussed in Section 4 of the main paper [26].

4.1 ML Enhancements in BB

4.1.1 Background on ML for Branch-and-Bound

ML methods are increasingly embedded into BB frameworks to enhance decision-making within different algorithmic steps. However, most learned policies generalize reliably only within a narrow family of instances, and their transfer across unrelated problem classes remains an open challenge (see also Section of the main paper [26]). In a generic setting, each node n_k in the search tree corresponds to a relaxation of the original problem, represented by features such as reduced costs, dual values, pseudo-costs, integrality gaps, and constraint activity statistics. An ML model, parameterized by θ, takes these features as input and produces scores or probabilities that guide solver actions. Formally, given feature vectors $\text{feat}(n_k)$ for node n_k, a trained model f_θ computes $y_k = f_\theta(\text{feat}(n_k))$, where y_k may represent the predicted quality of a relaxation, the likelihood of a constraint being active, or the effectiveness of a candidate cut or branching variable. Decisions are then made by selecting the best-scoring candidate, for example

$$n^* = \underset{n_k \in \mathcal{N}}{\operatorname{argmin}}\, y_k \quad \text{or} \quad j^* = \underset{j \in \mathcal{B}(\mathcal{S}_k)}{\operatorname{argmax}}\, y_j,$$

depending on whether the model is guiding node selection, branching, or cut ranking. Here, $\mathcal{B}(\mathcal{S}_k)$ denotes the set of fractional variables eligible for branching at node $\mathcal{S}_k$, that is,

$$\mathcal{B}(\mathcal{S}_k) = \{j \mid x_j^* \notin \mathbb{Z}\}$$

for the current LP relaxation solution x^* at node $\mathcal{S}_k$.

Supervised learning (SL) is a core paradigm in ML where a model is trained on input–output pairs to minimize a loss between predicted outputs and known labels. In the context of `BB`, examples include training a model to predict the branching decision (output) based on the features of a node (input) collected from expert demonstrations.

Imitation learning (IL) is a specific form of SL in which the model learns to mimic an expert's behaviour. Instead of receiving rewards as in RL, the learner is provided with demonstrations (state–action pairs) from an expert, such as strong branching decisions or oracle node selections, and learns a policy that imitates these expert actions.

Graph Neural Networks (GNNs) are neural network architectures designed to operate on graph-structured data. They iteratively update node representations by aggregating and transforming information from neighboring nodes and edge features, making them well-suited for representing `BB` subproblems as bipartite graphs.

Surrogate models are learned approximation models that replace expensive objective or constraint evaluations with fast predictions, enabling efficient bounding and pruning decisions.

Active set methods focus computation on a selected subset of variables or constraints deemed most relevant, reducing problem dimensionality while preserving correctness.

Decomposition models split a complex optimization problem into smaller, loosely coupled subproblems whose solutions are coordinated to obtain the overall optimum.

These models can also predict decompositions of large-scale MINLPs, determine surrogate approximations for expensive evaluations, and reduce problem dimensionality by predicting active constraints. As described in the following algorithms, GNNs are used to imitate strong branching, regression models predict relaxation quality, surrogate models approximate black-box objectives, supervised classifiers filter active constraints, decomposition models partition variables and constraints, and ranking networks select high-quality cutting planes. All these approaches share a common principle: they integrate learned patterns from data into `BB` sub-steps such as (S0_1 or S0_2), (S1b_1 or S1b_2), (S1c_1 or S1c_2), (S1e_1 or S1e_2), and (S3_1 or S3_2) of Algorithms 1 and 2 provided in the main paper [26], improving efficiency while preserving the exactness of the underlying solver.

4.1.2 Algorithm 2 – Branching decision prediction

Algorithm 2 [12] summarizes the learned branching procedure proposed by Gasse et al., where a graph neural network is trained to imitate strong branching decisions within a `BB` framework. This algorithm has five steps ($S1_2$)–($S5_2$). For each `BB` node n_k with fractional solution x^*, this algorithm performs these five steps. Here, n_k denotes the current node in the search tree with associated LP relaxation and fractional solution x^*.

In ($S1_2$), the subproblem at n_k is encoded as a bipartite graph $G = (V, C, E)$, where V is the set of variable nodes with feature vectors such as reduced costs and pseudo-costs, C is the set of constraint nodes with features such as slacks and dual values, and E is the set of edges linking variables and constraints with edge features $e_{i,j}$ representing the coefficient $A_{i,j}$ of variable j in constraint i.

In ($S2_2$), message passing on G updates node embeddings v_j and c_i of variable and constraint nodes according to neural update functions f_C and f_V:

$$c_i := f_C\Big(c_i, \sum\nolimits_{j:(i,j)\in E} g_C(c_i, v_j, e_{i,j})\Big), \tag{16}$$

$$v_j := f_V\Big(v_j, \sum\nolimits_{i:(i,j)\in E} g_V(c_i, v_j, e_{i,j})\Big), \tag{17}$$

where g_C and g_V are message functions that aggregate information from neighbouring nodes.

In ($S3_2$), a final multilayer perceptron is applied with a masked `softmax` over candidate variables:

$$\pi_\theta(a = j \mid s_t) := \texttt{softmax}(W v_j + b), \tag{18}$$

where the weights W and bias b of the perceptron produce logits for each candidate branching variable, and $\pi_\theta(a = j \mid s_t)$ is the probability that variable x_j is selected at state s_t.

In ($S4_2$), the branching variable

$$j^* := \operatorname*{argmax}_j \pi_\theta(a = j \mid s_t) \tag{19}$$

is selected, after which in ($S5_2$) child nodes are created by enforcing

$$x_{j^*} \le \lfloor x^*_{j^*} \rfloor \quad \text{and} \quad x_{j^*} \ge \lceil x^*_{j^*} \rceil. \tag{20}$$

This algorithm can be used to improve the branching step ($S3_{??}$ or $S3_{??}$) of the `IBB` and `MIBB` algorithms.

Algorithm 2 GNN-based Learned Branching

Input: MILP instance, trained GNN policy π_θ
for each `BB` node n_k with fractional solution x^* **do**
($S1_2$) **Graph encoding:** encode the subproblem at n_k as a bipartite graph $G = (V, C, E)$.
($S2_2$) **Message passing:** update node embeddings using (16) and (17).
($S3_2$) **Score computation:** apply a multilayer perceptron with masked `softmax` over candidate variables using (18).
($S4_2$) **Branch selection:** select the branching variable j^* by (19).
($S5_2$) **Branching:** branch on x_{j^*} by creating two child nodes as in (20).
end for

This learned strong-branching surrogate currently applies to MILP problems; its conceptual extension to spatial branching for MINLP follows the framework of Ghaddar et al. [13].

4.1.3 Algorithm 3 – Learning relaxation quality via neural branching

Algorithm 3 [36] summarizes the neural branching procedure through five steps ($S1_3$)–($S5_3$). For each `BB` node n_k, the method first encodes the current node and its LP relaxation into feature vectors $f(v)$ for all candidate branching variables ($S1_3$). These features are passed to the trained branching model π_θ to produce branching scores $\hat{p}(v)$ approximating the FSB preference ($S2_3$). The variable with the highest score, $v^* = \text{argmax}_v \hat{p}(v)$, is selected as the branching variable ($S3_3$). Branching is then performed on v^*, creating two child nodes with updated floor and ceiling bounds ($S4_3$). Finally, the generated child nodes are added to the `BB` queue for subsequent exploration ($S5_3$).

Algorithm 3 Neural Branching

Input: Active `BB` node queue, trained branching model π_θ
for each active `BB` node n_k popped from the queue **do**
($S1_3$) **Feature extraction:** extract feature vector $f(v)$ for each candidate branching variable v at node n_k.
($S2_3$) **Score computation:** evaluate the trained model, $\hat{p}(v) = \pi_\theta(f(v))$, for all v.
($S3_3$) **Variable selection:** select variable $v^* = \text{argmax}_v \hat{p}(v)$.
($S4_3$) **Branching:** branch on v^* to create two child nodes with updated floor and ceiling bounds.
($S5_3$) **Queue update:** add child nodes to the `BB` node queue.
end for

4.1.4 Algorithm 4 – Surrogate modelling of expensive evaluations

Algorithm 4 [29] summarizes the surrogate-based bounding procedure in five steps ($S1_4$)–($S5_4$). For each BB node n_k, in ($S1_4$), the algorithm first collects historical data from previously evaluated solutions to form a dataset $\mathcal{D}$. In ($S2_4$), the surrogate model $\hat{f}_\phi$ is trained or incrementally updated using $\mathcal{D}$. In ($S3_4$), an approximate bound $\hat{b}_k$ is computed by minimizing the surrogate model over the relaxation region $R(n_k)$, i.e.,

$$\hat{b}_k = \min_{x \in R(n_k)} \hat{f}_\phi(x). \tag{21}$$

In ($S4_4$), this surrogate-derived bound is used to guide pruning and branching decisions during the bounding phase. Finally, in ($S5_4$), the surrogate is periodically validated and refined by evaluating the original expensive function $f(x)$ at selected points, updating $\mathcal{D}$ and retraining $\hat{f}_\phi$.

Algorithm 4 reformulates their surrogate-update procedure within a BB bounding phase for illustrative purposes.

Algorithm 4 Surrogate-Based Bounding (adapted from Li et al., 2021)

Input: BB node n_k with expensive black-box objective $f(x)$, active node queue, and current surrogate model $\hat{f}_\phi$
Output: Approximate bound value $\hat{b}_k$
for each active BB node n_k popped from the queue **do**
($S1_4$) **Data collection:** gather historical samples $\mathcal{D} = \{(x^{(i)}, f(x^{(i)}))\}$ from previously evaluated solutions.
($S2_4$) **Surrogate update:** train or incrementally update the surrogate model $\hat{f}_\phi(x)$ using $\mathcal{D}$.
($S3_4$) **Approximate bounding:** compute the node's approximate bound by solving the relaxed subproblem with the surrogate (21).
($S4_4$) **Guided pruning:** use $\hat{b}_k$ as an approximate bound to guide pruning and branching decisions in BB.
($S5_4$) **Validation and refinement:** periodically evaluate $f(x)$ at selected points in $R(n_k)$, add these evaluations to $\mathcal{D}$, and update $\hat{f}_\phi$ for improved accuracy.
end for

This surrogate-assisted approach aligns with Li et al.'s broader goal of reducing evaluation costs in hierarchical and nonlinear optimization, and can be interpreted as a learning-augmented bounding mechanism within exact BB solvers.

4.1.5 Algorithm 5 – Learning cut selection

Algorithm 5 [23] summarizes the cut selection procedure in six steps ($S1_5$)–($S6_5$). For each candidate cut c_j, in ($S1_5$) the algorithm first extracts instance-specific features, and in ($S2_5$) evaluates the learned scoring function $s_j = r_\omega(\text{features}(c_j))$. In ($S3_5$) the candidate cuts are then ranked by predicted effectiveness, and in ($S4_5$) the top-ranked cuts are selected to form $\mathcal{C}^*$, respecting solver or resource constraints. These selected cuts are added to the node's LP/NLP relaxation in ($S5_5$). Finally, in ($S6_5$) the bounding phase continues with the enhanced relaxation, which improves pruning efficiency and overall solver performance.

Algorithm 5 Cut Ranking for Learning Cut Selection

Input: BB node n_k with candidate cut set $\mathcal{C}$, trained ranking model r_ω
Output: Selected subset of cuts $\mathcal{C}^*$
($S1_5$) **Feature extraction:** for each candidate cut $c_j \in \mathcal{C}$, extract instance-specific features.
($S2_5$) **Scoring:** compute a predicted effectiveness score $s_j = r_\omega(\text{features}(c_j))$ for each cut.
($S3_5$) **Ranking:** sort candidate cuts in descending order of scores s_j.
($S4_5$) **Cut selection:** select the top k cuts to form $\mathcal{C}^* = \{c_j \mid s_j \text{ in top-}k\}$.
($S5_5$) **Relaxation update:** add $\mathcal{C}^*$ to the LP/NLP relaxation at node n_k.
($S6_5$) **Bounding continuation:** continue the bounding phase with the updated relaxation.

4.1.6 Algorithm 6 – Learning variable activity

Algorithm 6 [41] summarizes the variable activity reduction procedure in five steps ($S1_6$)–($S5_6$). For each binary variable x_i, the algorithm first extracts descriptive features in ($S1_6$) and predicts its activity

$$a_i = g_\psi(\text{features}(x_i)), \tag{22}$$

where $a_i = 1$ means predicted active and $a_i = 0$ inactive, using the trained classifier g_ψ in ($S2_6$). After all predictions are made, the active variable set

$$X_{\text{active}} = \{x_i \in X \mid a_i = 1\} \tag{23}$$

is formed in ($S3_6$) and a reduced MIP is built using only these variables and their associated constraints in ($S4_6$). Finally, this reduced MIP is solved within the bounding phase in ($S5_6$) to accelerate the BB search while preserving feasibility and optimality guarantees.

Algorithm 6 Learning-Based Variable Activity Reduction

Input: MIP instance with binary variable set $X = \{x_1, \ldots, x_n\}$ and trained classifier g_ψ
Output: Reduced active variable set X_{active}
for each binary variable $x_i \in X$ **do**
($S1_6$) **Feature extraction:** compute variable-level features.
($S2_6$) **Prediction:** predict activity a_i by (22).
end for
($S3_6$) **Reduced set formation:** form X_{active} as (23).
($S4_6$) **Reduced MIP building:** build a smaller MIP using only variables in X_{active} and associated constraints.
($S5_6$) **Bounding:** solve the reduced MIP within the bounding phase to speed up the search while maintaining feasibility and optimality.

This pre-solve filtering is independent of branching heuristics and complements ML-enhanced branching or node-selection policies discussed in Sections 4.1 and 4.2.

4.1.7 Algorithm 7 – Learning decomposition strategies

Algorithm 7 [35] summarizes the learned decomposition strategy selection in five steps ($S1_7$)–($S5_7$). First, in ($S1_7$), the problem instance is encoded as a graph $G = (V, E, F)$ with nodes V for variables and constraints, edges E for couplings, and features F describing structural and functional properties. Next, in ($S2_7$), the GNN classifier h_η predicts whether to decompose or solve monolithically. Depending on the decision, in ($S3_7$), the solver selects either a decomposition-based algorithm such as `OA` or a monolithic method such as `BB`. Then, in ($S4_7$), `MIBB` is initialized with the chosen strategy. Finally, in ($S5_7$), the selected algorithm is executed, ensuring that the solver leverages the predicted best approach to reduce overall solution time.

Algorithm 7 Learning When to Decompose

Input: MINLP instance with variables X, constraints $\mathcal{C}$; trained graph classifier h_η
Output: Decision to solve monolithically or with decomposition
($S1_7$) **Graph encoding:** represent the problem as a graph G.
($S2_7$) **Classification:** evaluate $d = h_\eta(G) \in \{\text{decompose}, \text{monolithic}\}$.
($S3_7$) **Strategy selection:** if $d =$ decompose, select a decomposition algorithm (e.g., `OA`); otherwise, select a monolithic solver (e.g., `BB`).
($S4_7$) **Initialization:** initialize `MIBB` with the chosen solution strategy.
($S5_7$) **Execution:** solve the instance using the selected approach to exploit the predicted performance advantage.

This supervised decomposition-selection module complements other learning-enhanced solver components (e.g., branching and parameter tuning) by operating at the initialization level before the BB search begins, and can in principle be extended to other decomposition frameworks beyond OA.

4.2 RL Enhancements in BB

RL has emerged as a natural framework for learning sequential decision-making policies in BB algorithms. Notable RL-based approaches are discussed below.

4.2.1 Background on RL for BB

RL is an ML paradigm in which an agent interacts with an environment by taking actions in given states to maximize cumulative rewards over time. Unlike SL, RL does not rely on labeled input–output pairs; instead, the agent learns from feedback in the form of rewards.

RL provides a natural framework for optimizing sequential decisions in BB search. At iteration t, the solver is in state s_t, which may include the set of open nodes $\mathcal{N}_t$, the search history $\mathcal{H}_t$, the current incumbent bound UB_t, and solver parameters. An RL agent, represented by a policy $\pi_\theta(a \mid s_t)$ with trainable parameters θ, outputs a probability distribution over available actions a_t, such as selecting a node $\mathcal{S}_k$ for exploration, choosing a branching variable j^*, or updating algorithmic parameters λ_t. The agent receives feedback in the form of rewards r_t, for instance, improvements in bound gaps or reductions in search tree size, and updates θ using policy-gradient or value-based methods:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t \mid s_t)\big(r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)\big), \tag{24}$$

where α is the learning rate, V_θ is a learned value function, and $\gamma \in [0, 1]$ is the discount factor controlling the weight of future rewards (a smaller γ emphasizes immediate improvements such as node reduction, while values close to 1 encourage long-term strategies minimizing total search effort). Equation (24) represents a generic policy-gradient update (Actor–Critic form) used here for consistency; specific works such as [31,40] employ equivalent REINFORCE-style updates.

Q-learning is a value-based RL method that learns an action–value function $Q(s, a)$ estimating the expected cumulative reward of taking action a in state s. This function guides the selection of actions that maximize long-term reward.

Policy-gradient methods directly optimize the policy parameters θ to increase the likelihood of actions that lead to higher cumulative rewards, as demonstrated in the update equation above.

GNNs are neural architectures designed to operate on graph-structured data. They update node representations by aggregating information from neighboring nodes and edge features, making them well-suited for representing the structure of BB subproblems.

RL-based methods go beyond static heuristics by dynamically balancing exploration and exploitation as the search progresses. In the algorithms described later, Q-learning is used to prioritize nodes in step S1a, deep RL policies based on GNNs adaptively guide both node selection and branching in steps S1a and S3, parameter-control agents adjust solver hyperparameters during initialization and throughout S0, step-size controllers adjust updates within sub-solvers for S1b, and RL-based cut selectors improve the generation of cutting planes in S1c. Through these mechanisms, RL agents continuously interact with the BB process to enhance search efficiency and convergence.

4.2.2 Algorithm 8 – Node selection

Algorithm 8 [19] summarizes the imitation-learned node selection procedure in five steps ($S1_8$)–($S5_8$). First, for each node in the open list, a feature vector is extracted in ($S1_8$). Next, the learned linear scoring function computes a score for each node in ($S2_8$), and the node with the highest score is selected for expansion in ($S3_8$). The solver then expands the selected node and updates the open list in ($S4_8$). Finally, the scoring weights are refined via IL updates to better approximate the oracle's ranking policy in ($S5_8$).

Algorithm 8 Imitation-Learned Node Selection

Input: Open node set $\mathcal{N}$, learned scoring function w
Output: Selected node $\mathcal{S}_k$ to expand
($S1_8$) **Feature extraction:** For each $n_i \in \mathcal{N}$, extract feature vector $\phi(n_i)$.
($S2_8$) **Scoring:** Compute score $s_i = w^T \phi(n_i)$ for each node.
($S3_8$) **Node selection:** Select the node $\mathcal{S}_k = \text{argmax}_{n_i \in \mathcal{N}} s_i$.
($S4_8$) **Expansion:** Expand $\mathcal{S}_k$ and update the open list $\mathcal{N}$.
($S5_8$) **IL update:** Update w to better imitate the oracle's ranking policy.

4.2.3 Algorithm 9 – Cut selection

Algorithm 9 [40] summarizes the RL-based cut selection procedure in eight steps ($S1_9$)–($S8_9$). First, features are extracted for each candidate cut in ($S1_9$), and the solver encodes the state s_t along with the candidate cut set $\mathcal{C}_t$ in ($S2_9$). Next, the policy network selects a subset of cuts to add to the relaxation in ($S3_9$), forming the set $\mathcal{C}_t^*$ in ($S4_9$). The chosen cuts are added to the LP relaxation and the relaxation is re-solved in ($S5_9$). The solver observes a reward based on

the quality of the cuts, such as bound improvement or node reduction, in ($S6_9$), and the state is updated to s_{t+1} in ($S7_9$). If the algorithm is in training mode, the policy parameters θ are updated using a REINFORCE-style policy-gradient step (as in Tang et al., 2020, Algorithm 1) to maximize expected cumulative reward in ($S8_9$), following Eq. (24).

Algorithm 9 RL-based Cut Selection (Tang et al., 2020)

Input: Current LP relaxation state s_t, candidate cut set $\mathcal{C}_t$, trained policy $\pi_\theta(c_t \mid s_t)$ with parameters θ, learning rate $\alpha > 0$, discount factor $\gamma \in [0, 1]$, `training` (Boolean flag)
Output: Selected cut subset $\mathcal{C}_t^*$
for each cut selection step $t = 0, 1, 2, \ldots$ until convergence **do**
($S1_9$) **Feature extraction:** Extract features for each candidate cut $c_j \in \mathcal{C}_t$.
($S2_9$) **State encoding:** Encode s_t and $\mathcal{C}_t$ into a state representation.
($S3_9$) **Cut selection:** Use policy to select cuts $c_t \sim \pi_\theta(c_t \mid s_t)$.
($S4_9$) **Subset formation:** Form $\mathcal{C}_t^* = \{c_j$ chosen by policy$\}$.
($S5_9$) **Relaxation update:** Add $\mathcal{C}_t^*$ to the LP relaxation and re-solve.
($S6_9$) **Reward observation:** Observe reward r_t (e.g., bound improvement or node reduction).
($S7_9$) **State update:** Update state to s_{t+1}.
if `training` **then**
($S8_9$) **Policy gradient update:** Update θ by (24).
end if
end for

4.2.4 Algorithm 10 – Adaptive search strategies

Algorithm 10 [28] summarizes the adaptive node comparison procedure in six steps ($S1_{10}$)–($S6_{10}$). First, each candidate node is encoded as a bipartite graph with constraint, variable, and global features in ($S1_{10}$). Next, a GNN processes these graphs to generate scalar scores $s_i = g(n_i)$ in ($S2_{10}$). The model then computes a preference value $p = \sigma(s_1 - s_2)$ in ($S3_{10}$) and selects the preferred node based on whether $p > 0.5$ in ($S4_{10}$). The solver updates the open list ordering according to the learned comparison outcome in ($S5_{10}$) and repeats pairwise comparisons as necessary to perform node selection in ($S6_{10}$).

Algorithm 10 Adaptive Node Comparison with GNN (Labassi et al., 2022)

Input: Candidate nodes n_1, n_2 from open list; trained GNN scoring function $g(\cdot)$
Output: Preferred node between n_1 and n_2
($S1_{10}$) **Graph encoding:** Encode each node n_i as a bipartite graph with constraint, variable, and global features.
($S2_{10}$) **GNN scoring:** Apply the GNN to obtain scalar scores $s_i = g(n_i)$.
($S3_{10}$) **Preference computation:** Compute preference $p = \sigma(s_1 - s_2)$.
($S4_{10}$) **Node selection:** if $p > 0.5$, prefer n_2, else prefer n_1; end if
($S5_{10}$) **Open list update:** Update the open list ranking according to the comparison result.
($S6_{10}$) **Repeat comparison:** Repeat comparisons as needed to perform full node selection.

4.2.5 Algorithm 11 – Adaptive parameter control

Algorithm 11 [31] summarizes the adaptive parameter control procedure in six steps ($S0_{11}$)–($S5_{11}$). First, the solver is initialized with an initial state and parameter set in ($S0_{11}$). At each decision step, the current state s_t is observed in ($S1_{11}$), and the high-level policy outputs new parameters $\lambda_t = \pi_\theta(s_t)$ in ($S2_{11}$). The solver then applies λ_t and runs for a fixed horizon H, collecting a reward r_t that reflects performance in ($S3_{11}$). The state is updated to s_{t+1} in ($S4_{11}$). If training is active, the policy parameters θ are updated using an RL policy gradient rule to maximize expected rewards in ($S5_{11}$).

Algorithm 11 Adaptive Parameter Control with RL (Wang et al., 2025)

Input: Initial state s_0, initial parameters λ_0, policy $\pi_\theta(\lambda_t \mid s_t)$ with parameters θ, learning rate $\alpha > 0$, discount factor $\gamma \in [0, 1]$, `training` (Boolean flag)
Output: Adapted parameter sequence $\{\lambda_t\}$
($S0_{11}$) **Initialization:** Initialize solver with s_0 and λ_0.
for each step $t = 0, 1, 2, \ldots$ until termination **do**
  ($S1_{11}$) **State observation:** Observe current state s_t.
  ($S2_{11}$) **Parameter update:** Compute new parameters $\lambda_t = \pi_\theta(s_t)$.
  ($S3_{11}$) **Execution and reward:** Apply λ_t, run solver for horizon H, and observe reward r_t.
  ($S4_{11}$) **State update:** Update state to s_{t+1}.
  if `training` **then**
    ($S5_{11}$) **Policy gradient update:** Update θ by (24).
  end if
end for

References

[1] Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, James Luedtke, and Ashutosh Mahajan. Mixed-integer nonlinear optimization. *Acta Numerica*, 22:1–131, April 2013.

[2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[3] Samuel Burer and Anthony N. Letchford. Non-convex mixed-integer nonlinear programming: A survey. *Surveys in Operations Research and Management Science*, 17(2):97–106, 2012.

[4] Richard H. Byrd, Jorge Nocedal, and Richard A. Waltz. *KNITRO: An Integrated Package for Nonlinear Optimization*, pages 35–59. Springer US, 2006.

[5] Matteo Cacciola, Alexandre Forel, Antonio Frangioni, and Andrea Lodi. The differentiable feasibility pump. In *Proceedings of IPCO 2025*, pages 157–171, 2025.

[6] Kevin Cunningham and Linus Schrage. The LINGO algebraic modeling language. *Modeling languages in mathematical optimization*, pages 159–171, 2004.

[7] Arnaud Deza and Elias B Khalil. Machine learning for cutting planes in integer programming: A survey. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 6592–6600, 2023.

[8] Naveed Ejaz and Salimur Choudhury. A comprehensive survey of linear, integer, and mixed-integer programming approaches for optimizing resource allocation in 5g and beyond networks. *arXiv*, 2025. Reviewing LP, ILP, MILP with a focus on RL-based and hybrid heuristics.

[9] FICO. *FICO Xpress Optimization Suite*, 2021. https://www.fico.com/products/fico-xpress-optimization-suite.

[10] Christodoulos A. Floudas. *Deterministic Global Optimization*. Springer US, 2000.

[11] Robert Fourer, David M. Gay, and Brian W. Kernighan. AMPL: A mathematical programming language. In Stefan M. Stefanov, editor, *Algorithms and Model Formulations in Mathematical Programming*, volume 51 of *NATO ASI Series (Series F: Computer and Systems Sciences)*, pages 150–151. Springer, 1990.

[12] Maxime Gasse, Didier Chételat, Félix Ferroni, Andrea Lodi, and Giulia Zarpellon. Exact combinatorial optimization with graph convolutional neural networks. In *NeurIPS*, 2019.

[13] Bissan Ghaddar, Ignacio Gómez-Casares, Julio González-Díaz, Brais González-Rodríguez, Beatriz Pateiro-López, and Sofía Rodríguez-Ballesteros. Learning for spatial branching: An algorithm selection approach. *INFORMS Journal on Computing*, 35(5):1024–1043, 2023.

[14] Philip E Gill, Walter Murray, and Michael A Saunders. SNOPT: An sqp algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005.

[15] Ambros Gleixner, Gerald Gamrath, Thorsten Koch, Matthias Miltenberger, Ted Ralphs, Domenico Salvagnin, Yuji Shinano, Dieter Weninger, Timo Berthold, and Tobias Achterberg. SCIP optimization suite. https://scipopt.org. Accessed: 2025-01-01.

[16] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*, 2023. https://www.gurobi.com.

[17] Gurobi Optimization, LLC. *Gurobi Instant Cloud Guide*. Gurobi Optimization, LLC, Beaverton, OR, USA, revision 3de125aa4 edition, June 2025. Available at https://www.gurobi.com/documentation/.

[18] Gurobi Optimization, LLC. *Gurobi Remote Services Guide, Version 12.0*. Gurobi Optimization, LLC, Beaverton, OR, USA, revision 1f1d9c738 edition, September 2025. Available at https://www.gurobi.com/documentation/.

[19] He He, Hal Daumé III, and Jason Eisner. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing Systems*, pages 3293–3301, 2014.

[20] Reiner Horst and Panos M Pardalos. *Handbook of global optimization*, volume 2. Springer Science & Business Media, 2013.

[21] Reiner Horst and Hoang Tuy. *Global Optimization: Deterministic Approaches*. Springer, 1996.

[22] Lingying Huang, Xiaomeng Chen, Wei Huo, Jiazheng Wang, Fan Zhang, Bo Bai, and Ling Shi. Branch and bound in mixed integer linear programming problems: A survey of techniques and trends. *arXiv preprint arXiv:2111.06257*, 2021.

[23] Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123:108353, March 2022.

[24] IBM. *IBM ILOG CPLEX Optimization Studio*, 2021. https://www.ibm.com/products/ilog-cplex-optimization-studio.

[25] Surbhi Bhatia Khan, Reena Dadhich, and Deepali Sharma. A survey on non-linear optimization and global optimization methods. *International Journal of Advanced Research in Science, Communication and Technology*, 9(1), 2024.

[26] Morteza Kimiaei, Vyacheslav Kungurtsev, and Brian Olimba. Machine learning algorithms for improving exact classical solvers in mixed integer continuous optimization, 2025.

[27] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4475–4482, 2021.

[28] Abdel Ghani Labassi, Didier Chételat, and Andrea Lodi. Learning to compare nodes in branch and bound with graph neural networks. In *Advances in Neural Information Processing Systems*, volume 35, pages 22891–22904, 2022.

[29] Zhongguo Li, Zhen Dong, Zhongchao Liang, and Zhengtao Ding. Surrogate-based distributed optimisation for expensive black-box functions. *Automatica*, 125:109407, March 2021.

[30] Robin Lougee-Heimer. The common optimization interface for operations research: Coin-OR. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[31] Wangtao Lu, Yufei Wei, Jiadong Xu, Wenhao Jia, Liang Li, Rong Xiong, and Yue Wang. Reinforcement learning for adaptive planner parameter tuning: A perspective on hierarchical architecture. *arXiv preprint arXiv:2503.18366*, 2025.

[32] Makhorin, Andrew. *GNU Linear Programming Kit, Version 5.0*, 2021. https://www.gnu.org/software/glpk/.

[33] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey, October 2021.

[34] Ruth Misener and Christodoulos A. Floudas. ANTIGONE: Algorithms for continuous / integer global optimization of nonlinear equations. *Journal of Global Optimization*, 59(2–3):503–526, March 2014.

[35] Ilias Mitrai and Prodromos Daoutidis. Taking the human out of decomposition-based optimization via artificial intelligence, part i: Learning when to decompose. *Computers & Chemical Engineering*, 186:108688, July 2024.

[36] Venkatesh Nair, Annemarie Plaat, Ozan Gunluk, and Willem-Jan Van Hoeve. Solving mixed-integer programs using neural networks. In *NeurIPS*, 2020.

[37] Lara Scavuzzo, Karen Aardal, Andrea Lodi, and Neil Yorke-Smith. Machine learning augmented branch and bound for mixed integer linear programming. *Mathematical Programming, Series B*, 2024.

[38] Olivier Sigaud and Freek Stulp. Policy search in continuous action domains: an overview. *Neural Networks*, 113:28–40, 2019.

[39] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE Transactions on Cybernetics*, 50(8):3668–3681, August 2020.

[40] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning*, pages 9367–9376. PMLR, 2020.

[41] Niki Triantafyllou and Maria M. Papathanasiou. Deep learning enhanced mixed integer optimization: Learning to reduce model dimensionality. *Computers & Chemical Engineering*, 187:108725, August 2024.

[42] Victor Uc-Cetina, Nicolás Navarro-Guerrero, Anabel Martin-Gonzalez, Cornelius Weber, and Stefan Wermter. Survey on reinforcement learning for language processing. *Artificial Intelligence Review*, 56(2):1543–1575, 2023.

[43] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[44] Yunhao Yang and Andrew Whinston. A survey on reinforcement learning for combinatorial optimization. In *2023 IEEE World Conference on Applied Intelligence and Computing (AIC)*, page 131–136. IEEE, July 2023.

[45] Jiayi Zhang, Chang Liu, Xijun Li, Hui-Ling Zhen, Mingxuan Yuan, Yawen Li, and Junchi Yan. A survey for solving mixed integer programming via machine learning. *Neurocomputing*, 519:205–217, 2023.

[46] Yi Zhang and Nikolaos V. Sahinidis. Solving continuous and discrete nonlinear programs with baron. *Computational Optimization and Applications*, December 2024.

[47] Yi Zhang and Nikolaos V. Sahinidis. Learning to deactivate probing with graph convolutional network for mixed-integer nonlinear programming. *Optimization Letters*, June 2025.