POLICY-CONDITIONED JOINT LEARNING FOR NODE SELECTION AND VARIABLE BRANCHING IN BRANCH-AND-BOUND

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern solvers for mixed-integer linear programs (MILPs) rely on branch-and-bound, whose efficiency depends largely on two recurring decisions, namely node selection and variable branching. While machine learning has been applied to both decisions, most existing methods learn them separately and overlook their interdependence. To address this issue, we propose a Policy-Conditioned Joint Learning (PCJL) framework, which learns node selection and variable branching as a coupled pair. PCJL encodes the variable–constraint graph of each instance once at the root and reuses this encoding at every branching decision. PCJL then couples two value networks, a branching network that ranks candidate variables in the context of the selected node and a node network that evaluates each candidate node by the cost measured under the action of this learned branching policy. Node targets are thus conditioned on the branching actions taken at deployment and measured under a common reference continuation. To adapt the depth of each descent to the state of the search, PCJL further overrides the fixed plunging rule whenever a gradient-boosted regressor predicts a cheaper depth. Experiments on four NP-hard benchmark families show that PCJL is the fastest of the compared methods on every family and reduces the solving time of SCIP by 15% to 63% on three of them. Moreover, its advantage carries over to larger instances on two families, where the reduction remains 28% to 42%.

1 INTRODUCTION

Mixed-integer linear programs (MILPs) model optimization problems in production planning, vehicle routing, and process scheduling (Wu et al., 2013; Schouwenaars et al., 2001; Floudas & Lin, 2005). Branch-and-bound solves an MILP by relaxing integrality, branching on subproblems with fractional solutions, and pruning subproblems whose relaxations cannot improve the incumbent (Land & Doig, 1960). At each iteration, node selection chooses the next subproblem, and variable branching chooses the fractional variable that splits it. Both decisions are traditionally delegated to hand-designed rules. Such rules rank open nodes by an estimate of their subtrees and fractional variables by past bound improvements. Each rule addresses one decision and takes the other as given.

However, neither decision can be made in isolation, because each restricts the choices of the other. The selected node determines which linear programming (LP) relaxation is solved next. Its solution determines which variables are fractional and thus eligible for branching. Conversely, the branching variable determines which children enter the frontier. Because the two decisions alternate throughout the search, a change to either decision propagates through all subsequent iterations. Different selection orders thus produce different search trees, as illustrated in Figure 1. In particular, the cost of selecting a node depends on the branching decision made at that node.

Hand-designed rules score each option by fixed local criteria and therefore ignore these downstream effects. Machine learning offers an alternative, because instances from the same application share structure that a policy trained on previously solved instances can exploit. For variable branching, policies have been learned by imitating strong branching (Alvarez et al., 2017; Gasse et al., 2019; Gupta et al., 2020) or by reinforcement learning from the search itself (Scavuzzo et al., 2022; Parsonson et al., 2023). For node selection, policies have been learned from oracle paths, pairwise

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

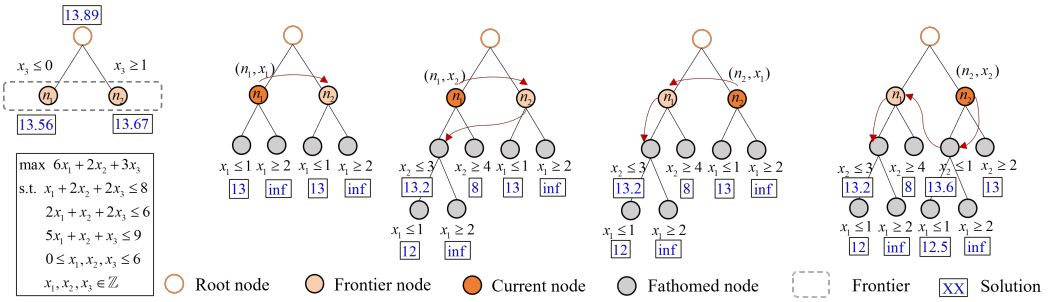

Figure 1: Interaction of the two decisions on a small MILP. Left: the root relaxation and the two open nodes after the first branching. Right: four continuations that select node n_1 or n_2 first and branch on a different variable at the selected node. Each combination produces a different tree, so the cost of selecting a node depends on the branching decision made at that node.

node comparisons, or delayed rewards (He et al., 2014; Labassi et al., 2022; Mattick & Mutschler, 2024; Zhang et al., 2025). A recent study learns both decisions within a single model (Du et al., 2025). However, each decision still imitates its own expert. The node target is therefore not measured under the branching actions that the learned model takes.

Learning the two decisions jointly is difficult because their training signals are coupled. The cost of a branching decision can be measured directly at its node. However, the cost of selecting a node is defined for a given branching decision at that node. Node costs should therefore be measured under the branching policy that will be deployed. This requires the branching policy to be fixed first. Each cost also depends on the continuation, because a single changed decision can alter later decisions.

To address these challenges, we propose Policy-Conditioned Joint Learning (PCJL). PCJL learns the branching policy first and then measures every node target under the actions of this fixed policy followed by a common reference continuation. The node targets are thus conditioned on the branching actions deployed with them. This paper makes three contributions. First, it encodes the variable–constraint graph of each instance once at the root, describes the current node and its branching candidates by lightweight descriptors, and computes the context of each node with a node encoder shared by the two networks. Second, a branching network ranks candidate variables in the context of the selected node, and a node network evaluates each candidate node by the cost measured under this learned branching policy. Both networks are fitted to LP work measured by replaying recorded searches under a common continuation. The node and branching policies depart from the reference policy only where their predicted advantage exceeds a calibrated threshold. Finally, a gradient-boosted regressor scores each candidate depth and overrides the fixed plunging rule whenever it predicts a cheaper one. Its targets are measured by the same replay construction under the frozen pair. PCJL is integrated into the solver SCIP (Bestuzheva et al., 2021). On four NP-hard benchmark families, it is the fastest of the compared methods on every family and reduces the solving time of SCIP by 15% to 63% on three of them. In contrast, no learned baseline is faster than SCIP on any family. On larger instances, the advantage of PCJL persists on two families.

2 Related Work

Most learned branching policies imitate strong branching. This expert solves two child relaxations per candidate and is therefore expensive. Alvarez et al. (2017) regress its scores with extremely randomized trees, Khalil et al. (2016) and Hansknecht et al. (2018) learn a pairwise ranking of the candidates, and Gasse et al. (2019) imitate its choice with a graph convolutional network. Zarpellon et al. (2021) add descriptors of the search tree to the candidate features. Other studies learn branching from the search itself with a tree Markov decision process (Scavuzzo et al., 2022) or retrospective trajectories (Parsonson et al., 2023). Kuang et al. (2024) learn symbolic rules that match GPU-based policies on a single CPU core. All of these methods leave node selection to the solver.

Node selection has been learned mainly from demonstrations of good search paths. He et al. (2014) imitate an oracle that expands the nodes on the path to the optimal solution, Song et al. (2018) derive demonstrations from the rollouts of the policy itself, and Yilmaz & Yorke-Smith (2021) learn which child to expand next. Labassi et al. (2022) compare pairs of open nodes with a Siamese graph neural network (GNN) trained to imitate a diving oracle. More recent work uses reinforcement learning and represents the state by the whole search tree (Mattick & Mutschler, 2024) or by a tripartite graph with leaf vertices (Zhang et al., 2025). In all of these methods, the label or reward of a node is obtained under the branching rule of the solver. Few studies learn both decisions. Du et al. (2025) learn node selection and variable branching through a shared graph representation and multi-task imitation. However, each task imitates its own expert, so the node target ignores the branching decisions of the learned model.

The state representation also determines the inference cost. Gasse et al. (2019) evaluate their GNN at every branching node. Gupta et al. (2020) evaluate a GNN only at the root and use a multilayer perceptron at the remaining nodes. Their policy therefore runs on a CPU. For node selection, Labassi et al. (2022) encode each compared node as a separate graph, and Zhang et al. (2025) rebuild a tripartite graph at every node selection. PCJL adopts the root-level design of Gupta et al. (2020) for variable branching and adds a node encoder shared by both networks.

Solvers also decide how far to follow a newly opened path before returning to the frontier. SCIP makes this choice with a fixed plunging rule (Bestuzheva et al., 2021). Learned diving (Nair et al., 2020) and heuristic scheduling (Chmiela et al., 2021) concern primal heuristics. In the main search, a fixed rule still sets the depth of each descent.

In summary, most existing methods learn one decision while holding the other fixed. Three issues remain open. Encoding the search state at every decision is costly, node targets are not aligned with the branching action of the learned model, and the depth of each descent follows a fixed rule. Section 4 addresses these issues.

3 Preliminaries and Problem Setting

3.1 Mixed-Integer Linear Programming

An MILP is an optimization problem with a linear objective, linear constraints, and integrality requirements on a subset of its variables. Stated as a maximization, it takes the form

$$\max_{x \in \mathbb{R}^p} \left\{ c^\top x : Ax \le b,\ l \le x \le u,\ x_i \in \mathbb{Z}\ \forall i \in \mathcal{I} \right\}, \tag{1}$$

where $c \in \mathbb{R}^p$ contains the objective coefficients, $A \in \mathbb{R}^{m \times p}$ and $b \in \mathbb{R}^m$ define the constraints, l and u bound the variables, and $\mathcal{I} \subseteq \{1, \dots, p\}$ indexes the integer variables. Each tuple $(c, A, b, l, u, \mathcal{I})$ defines an instance $\mathcal{M}$. Minimization is treated analogously.

An instance $\mathcal{M}$ can be represented as a variable–constraint bipartite graph $\mathcal{G} = (\mathcal{V} \cup \mathcal{C}, \mathcal{E})$ (Gasse et al., 2019; Nair et al., 2020). The sets $\mathcal{V}$ and $\mathcal{C}$ contain one vertex per variable and one vertex per constraint, respectively. An edge connects a variable vertex to a constraint vertex if the variable has a nonzero coefficient in that constraint. Vertex features describe the variables and the constraints, and edge features contain the corresponding coefficients of A.

3.2 Branch-and-Bound and the Order of Decisions

Branch-and-bound solves problem (1) by recursively partitioning it into subproblems that constitute the nodes of a search tree (Land & Doig, 1960). The unprocessed nodes are called open and form the frontier, and the best feasible solution found so far is called the incumbent. Processing a node applies domain propagation and solves its LP relaxation. The node is then closed if the relaxation is infeasible, if its optimal value does not exceed the objective value of the incumbent, or if its optimal solution satisfies the integrality constraints. Otherwise, branching on a fractional variable partitions the node into two children that enter the frontier.

Two decisions recur throughout this process. Node selection chooses the open node to be processed next, and variable branching chooses the fractional variable that splits a processed node. Let S denote

the search state before node selection and $\mathcal{A}_N(S)$ the set of open nodes at S. Selecting $n \in \mathcal{A}_N(S)$ and processing it produces the processed state

$$H_n = T_{\text{solver}}(S, n), \tag{2}$$

where T_{solver} represents propagation, the solution of the LP relaxation, and the remaining operations that precede branching. The branching candidates $\mathcal{A}_V(H_n)$ and their current LP features are determined by H_n and are not available at S. To cover nodes that are closed during processing, we define the extended candidate set

$$\overline{\mathcal{A}_V}(H_n) = \begin{cases} \mathcal{A}_V(H_n), & \text{if branching is required,} \\ \{\perp\}, & \text{if processing closes the node,} \end{cases} \tag{3}$$

where $\perp$ records that no branching action is taken.

Because the branching candidates are produced by processing, node selection and variable branching are nested. Enumerating all node–variable pairs at S would require processing every open node. A practical policy therefore selects a node before its branching candidates are known. Given a context h_n of node n that is available before processing, a policy that respects this order factorizes as

$$\pi_J(n, v \mid S) = \pi_N(n \mid S)\, \pi_V(v \mid H_n, h_n). \tag{4}$$

Equation (4) shows that the two decisions are made with different information. Variable branching observes the context h_n together with the candidates in H_n and their LP features. Node selection observes only the open nodes in S and their contexts. The decision with less information comes first.

After a node has been processed, the next selection can take a child or sibling of that node instead of a node elsewhere in the frontier. A sequence of such selections is called a descent, and branch-and-bound solvers usually limit its depth by a fixed plunging rule. In this paper, this depth becomes a third decision that is chosen at the start of each descent from $\mathcal{D} \cup \{\varnothing\}$, where $\mathcal{D}$ is a finite set of depths. The choice $d = 0$ ends the descent at once. The choice $\varnothing$ leaves the reference plunging rule in control. The next d selections then replace node selection in Equation (4). They are restricted to the children and siblings of the last processed node and leave π_V unchanged. Control returns to the full frontier after these selections or earlier if no such node is open.

3.3 Learning Objective and the Cost of a Decision

We seek node, branching, and depth policies π_N, π_V, and π_D that minimize the expected LP work of solving instances drawn from a distribution $\mathbb{P}$,

$$\min_{\pi_N, \pi_V, \pi_D} \mathbb{E}_{\mathcal{M} \sim \mathbb{P}} \left[W(\mathcal{M}; \pi_N, \pi_V, \pi_D) \right], \tag{5}$$

where the LP work W counts the simplex iterations performed in a complete solve of $\mathcal{M}$. The LP work of a solve results from many interacting decisions and does not reveal the contribution of any individual decision. We therefore evaluate each decision by the LP work of the subsequent search under a continuation, a fixed policy that makes all later decisions.

For node and branching costs, the continuation ρ is a fixed reference policy $\pi^0 = (\pi_N^0, \pi_V^0)$. Its node component selects the open node whose subtree has the best estimated optimal value and limits each descent by a fixed plunging rule. Its branching component is a frozen network pretrained to imitate full strong branching. We use this network because it provides a fixed branching map that every replay can query. We write $n^0 = \pi_N^0(S)$ for the reference node and $\beta^0(S, n)$ for the reference branching action at node n (Appendix B).

For a continuation ρ, let $W_\rho(S; n, v)$ denote the LP work performed from S onward when n is selected, v is applied if branching is required, and control then returns to ρ. Assigning a cost to a node requires a branching map β that supplies the action taken at the selected node. The reference policy defines one such map, namely β^0. At a node that is closed during processing, we set $\beta(S, n) = \perp$. The LP work incurred by selecting node n is then

$$W_\rho\big(S; n, \beta(S, n)\big). \tag{6}$$

The action of β at the node and the continuation ρ both determine the cost of the node.

The three costs are defined under different conditions. The cost of a branching action can be measured directly at the node where it is taken, because all candidates are available there and no learned

decision enters the measurement. The cost of a node is defined only for a fixed β. Different maps can even order the same open nodes differently (Proposition 1). The cost of a descent depth is defined only for a fixed node–branching pair.

Under this definition, a node cost is valid only for the branching map under which it was measured. A node cost measured with $\beta = \beta^0$ equals the cost under the learned branching policy only at nodes where the two policies select the same action. A node cost measured with a learned β requires the learned branching policy to be fixed first because any later change to it invalidates the records already collected. Branching costs are therefore measured first, node costs next under a fixed β, and depth costs last under the fixed node–branching pair. For node selection and variable branching, this order is the reverse of their execution order. Section 4 constructs the three policies under this constraint.

4 Policy-Conditioned Joint Learning

PCJL fits the three policies of Section 3.3 in their measurement order, namely the branching network first, the node network second, and the depth regressor last. The two networks predict the costs of branching and node selection decisions and share a node encoder (Figure 2). Only the branching network also reads a representation of the instance that is computed once at the root.

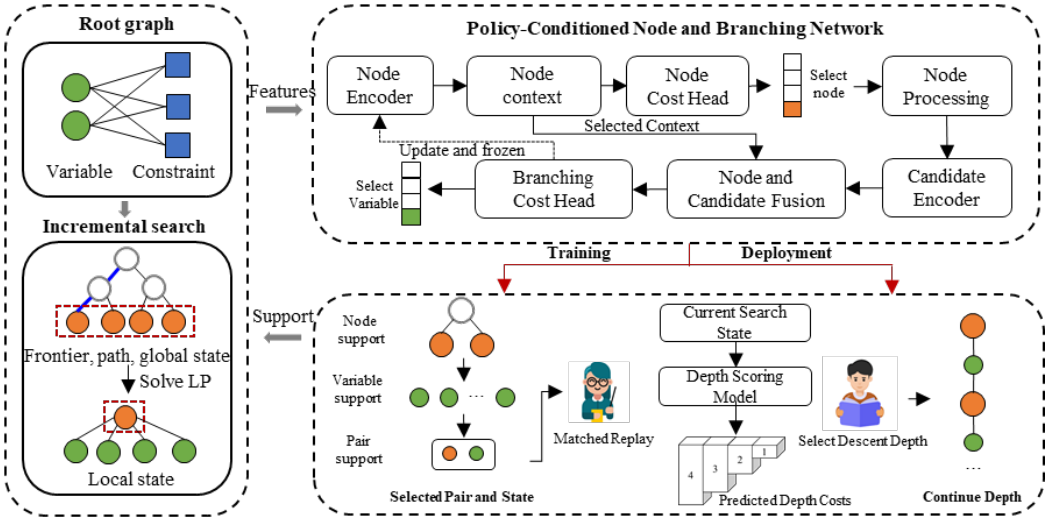

Figure 2: Architecture of PCJL. Both networks share the node encoder. Only the branching network reads the cached root representation. Node targets are measured under the actions of the frozen branching network, and the depth regressor is fitted last.

4.1 Representation

As in the hybrid design of Gupta et al. (2020), the graph of the instance is encoded only once. After the root LP relaxation has been solved, a graph encoder maps the graph $\mathcal{G}_{\text{root}}$ of the presolved instance (Section 3.1) to the variable embeddings

$$e_v = [\text{GNN}_\phi(\mathcal{G}_{\text{root}})]_v \in \mathbb{R}^{d_e}, \tag{7}$$

where ϕ denotes the encoder parameters and d_e the embedding dimension. Later branching calls reuse these embeddings without repeating message passing. The cache is built when the branching rule is first invoked at the root. The encoder is the GNN of the reference branching policy. It remains frozen because it belongs to the reference policy under which all costs are measured (Section 3.3). Search-dependent quantities enter through two descriptors (Table 4). The node descriptor $r_N(S, n)$ describes an open node n relative to the frontier. The candidate descriptor $r_V(H_n, v)$ contains the LP features, local-structure features, and reference rank of candidate v.

4.2 LEARNING THE NODE–BRANCHING PAIR

Continuation cost. Each cost compares two matched runs from a recorded state S. One run applies the tested action and follows the continuation ρ, and the other follows ρ alone.

Definition 1 (Continuation cost)**.** For a state S, a node $n \in \mathcal{A}_N(S)$, and an action $v \in \overline{\mathcal{A}_V}(H_n)$ for which both matched runs complete,

$$q_\rho(S, n, v) = \log \frac{W_\rho(S; n, v) + 1}{W_\rho(S; \varnothing) + 1}, \tag{8}$$

where $W_\rho(S; \varnothing)$ is the LP work without intervention. When $v = \perp$, only the node is forced.

Both policies choose within $\mathcal{C}_N(S) \subseteq \mathcal{A}_N(S)$ and $\mathcal{C}_V(H_n) \subseteq \mathcal{A}_V(H_n)$. A state is contested when its two best node estimates differ by at most 10^{-3} in relative terms. $\mathcal{C}_N(S)$ then contains the five best open nodes by estimate, and otherwise only n^0. $\mathcal{C}_V(H_n)$ contains the five best candidates by reference rank.

Branching network. Before node n is processed, the node encoder E_N computes its context

$$h_n = E_N\left(\mathrm{LayerNorm}(r_N(S, n))\right). \tag{9}$$

At a node that requires branching, a candidate encoder E_V combines the cached embedding of each candidate with its descriptor,

$$z_{n,v} = E_V\left([e_v\,; \mathrm{LayerNorm}(r_V(H_n, v))]\right), \tag{10}$$

where $[\,;\,]$ denotes concatenation. A linear map P projects the context to $u_n = Ph_n$, and the branching network predicts the cost of each candidate from the interaction of $z_{n,v}$ and u_n,

$$\psi_{n,v} = [z_{n,v}\,; u_n\,; z_{n,v} \odot u_n], \tag{11}$$

$$\hat{b}(S, n, v) = f_V(\psi_{n,v}), \qquad \widetilde{\beta}_F(S, n) \in \operatorname*{arg\,min}_{v \in \mathcal{C}_V(H_n)} \hat{b}(S, n, v), \tag{12}$$

where $\odot$ denotes elementwise multiplication and f_V is a multilayer perceptron with a scalar output. Through the elementwise product, the node context reweights the features of each candidate. The same candidate can therefore be ranked differently at different nodes. The deployed branching map keeps the reference action unless the predicted advantage of $\widetilde{\beta}_F(S, n)$ reaches a threshold γ_V,

$$\hat{\beta}_F(S, n) = \begin{cases} \widetilde{\beta}_F(S, n), & \hat{b}\big(S, n, \beta^0(S, n)\big) - \hat{b}\big(S, n, \widetilde{\beta}_F(S, n)\big) \geq \gamma_V, \\ \beta^0(S, n), & \text{otherwise.} \end{cases} \tag{13}$$

The parameters $\theta_B = (E_N, P, E_V, f_V)$ are fitted to the measured branching costs and then frozen. The threshold γ_V is calibrated on validation states after this freeze (Appendix B.2). The subscript F of $\hat{\beta}_F$ indicates that the map uses these frozen parameters and this threshold.

Node network. The node network places a scalar head f_N on the frozen node encoder,

$$\hat{g}(S, n) = f_N(h_n), \qquad \hat{n} \in \operatorname*{arg\,min}_{n \in \mathcal{C}_N(S)} \hat{g}(S, n). \tag{14}$$

Its target is the continuation cost of a node under the action of the frozen branching policy,

$$g_F(S, n) = q_\rho\big(S, n, \hat{\beta}_F(S, n)\big). \tag{15}$$

Whereas the reference target $g^0(S, n) = q_\rho(S, n, \beta^0(S, n))$ evaluates a node under the reference branching action, g_F evaluates it under the action that PCJL takes. A node target is policy-conditioned on a branching map if it evaluates every retained record under the action of that map. Only g_F is policy-conditioned on the deployed map $\hat{\beta}_F$. Both targets follow the reference continuation ρ after the branching action at the node. The node network is fitted to g_F with a consistency term $\lambda_C\big(\hat{g}(S, n) - \hat{b}(S, n, \hat{\beta}_F(S, n))\big)^2$ that penalizes disagreement between the two predictions of the same cost (Appendix B.2). The term calibrates the node head against the branching prediction during training and adds no information at selection time. Once f_N has been fitted, the node–branching pair is frozen. The learned pair follows the sequential factorization of Equation (4), and its node targets are defined under the frozen branching map $\hat{\beta}_F$. The shared context and the consistency term are only implementation choices. No gradient passes through the solver between the two decisions. Proposition 2 in Appendix A bounds the one-step loss of the pair by the two prediction errors and the branching threshold.

4.3 Descent Depth under the Frozen Pair

The depth policy decides at the start of each descent whether to override the reference plunging rule and which depth to apply (Section 3.2). A gradient-boosted regressor f_D takes a descriptor $r_D(S)$ of the search state (Table 4) and a candidate depth d as input and predicts the cost of that depth,

$$s_d(S) = f_D\big(r_D(S), d\big), \qquad d^\star \in \arg\min_{d \in \mathcal{D}} s_d(S), \qquad \hat{a} = \begin{cases} d^\star, & s_{d^\star}(S) < 0, \\ \varnothing, & \text{otherwise}, \end{cases} \tag{16}$$

where the action $\varnothing$ makes no intervention and leaves the descent to the reference plunging rule. A depth is applied only when its predicted cost is negative, and a tie at zero favors $\varnothing$.

Depth costs compare two matched runs from a recorded state S. The tested run applies depth d at S and then follows the frozen pair. The baseline run makes no intervention and therefore keeps the reference plunging rule. This comparison gives the target

$$q_{\mathcal{D}}(S, d) = \log \frac{W(S; d) + 1}{W(S; \varnothing) + 1}, \tag{17}$$

where $W(S; d)$ and $W(S; \varnothing)$ are the LP work of the two runs. Even the choice $d = 0$ differs from the baseline and has a cost. The regressor is fitted by squared error (Appendix B.2).

Deployment. During the solve, the three policies act in their execution order (Section 3.2). Outside descents, the node policy selects $\hat{n}$ by Equation (14). It replaces the reference node n^0 by $\hat{n}$ only when the predicted advantage $\hat{g}(S, n^0) - \hat{g}(S, \hat{n})$ exceeds a threshold γ_N calibrated on validation states (Appendix B.2). After the solver has processed a selected node n, the branching policy applies $\hat{\beta}_F(S, n)$ of Equation (13) and thus returns to the reference action when the predicted advantage is below γ_V. At the start of each descent, the depth policy chooses $\hat{a}$ by Equation (16) and keeps the reference plunging rule when $\hat{a} = \varnothing$. Because feasibility checks, bounding, and pruning are left to the solver, its optimality certificates remain valid. Algorithm 2 specifies the bookkeeping.

5 Computational Results

5.1 Experimental Setup

Benchmarks. We evaluate PCJL on four NP-hard MILP families used in prior work (Gasse et al., 2019; Labassi et al., 2022; Zhang et al., 2025), namely fixed-charge multicommodity network flow (FCMCNF) (Hewitt et al., 2010), maximum satisfiability (MaxSAT) (Ansótegui & Gabàs, 2017), generalized independent set (GISP) (Colombi et al., 2017), and combinatorial auctions (CA) (Leyton-Brown et al., 2000). For each family, a training pool supplies the replayed costs, and a validation pool serves model selection and the calibration of the two thresholds. Two evaluation pools of forty instances per family are generated with the same generators, a pool D for the main comparison and a pool E with larger instances. None of the evaluation instances is used for training, model selection, or calibration. For FCMCNF and GISP, the training instances are smaller than those of D. Pool D therefore already tests size generalization for these two families. All experiments use SCIP 8.0.3 (Bestuzheva et al., 2021) through PySCIPOpt (Maher et al., 2016) with default parameters apart from the replaced rules. Appendix C gives the generators and the pool sizes.

Baselines. We compare against the default configuration of SCIP with best-estimate node selection, plunging, and reliability pseudo-cost branching, and against three learned policies. GNN-Node is the node selection policy of Labassi et al. (2022), GNN-Branch is the branching policy of Gasse et al. (2019), and GNN-Joint is the joint model of Du et al. (2025). The learned baselines are trained as in their original papers on instances generated separately from the evaluation pools. GNN-Branch and GNN-Joint rebuild their graph input at every call. PCJL additionally uses costs measured by replay on its training pool. The comparison therefore concerns complete systems rather than methods with identical training budgets.

Training and evaluation. PCJL is trained with the staged procedure of Algorithm 1 under the configuration of Appendix B. Both thresholds and the training seed are selected on the validation pool and fixed before testing. Each test instance is solved with three solver seeds under a time limit of 300 seconds. All learned components run on the CPU (Appendix D).

Table 1: Performance on D. T is the solving time in seconds, L the number of LP iterations, and O the percentage of runs proving optimality. T and L are geometric means over seeds and instances, and the smallest recorded T and L of each family are in bold. The recorded values of T and L are interpreted jointly with O whenever some runs reach the time limit.

	FCMCNF			MaxSAT			GISP			CA		
Policy	T	L	O	T	L	O	T	L	O	T	L	O
SCIP	19.64	131658	100%	8.18	158916	100%	31.23	260288	100%	2.26	40504	100%
GNN-Node	20.06	128090	100%	8.50	155012	100%	33.53	271438	100%	2.75	45862	100%
GNN-Branch	19.72	**57206**	100%	21.72	147166	98%	93.00	**129168**	100%	6.51	14679	100%
GNN-Joint	37.13	89825	96%	23.23	76805	100%	172.35	185840	90%	4.87	17857	100%
PCJL	**12.62**	58360	100%	**3.02**	**45026**	100%	**30.20**	159319	100%	**1.93**	**11825**	100%

Table 2: Cumulative analysis on D. Each row above the rule introduces one cumulative change to the row above it. The two rows below the rule are controls: PCJL-2 with node targets g^0 instead of g_F, and PCJL with a constant depth instead of the learned depth. Metrics are as in Table 1, and the smallest recorded T and L of each family are in bold.

	FCMCNF			MaxSAT			GISP			CA		
Configuration	T	L	O	T	L	O	T	L	O	T	L	O
SCIP	19.64	131658	100%	8.18	158916	100%	31.23	260288	100%	2.26	40504	100%
PCJL-ref	15.80	93975	100%	6.93	103047	100%	32.33	259203	100%	2.15	22282	100%
PCJL-B	14.48	77376	100%	5.76	92328	100%	**30.09**	**156471**	100%	1.98	12031	100%
PCJL-2	13.05	65540	100%	3.42	56222	100%	31.27	158968	100%	1.96	11907	100%
PCJL	**12.62**	**58360**	100%	**3.02**	**45026**	100%	30.20	159319	100%	**1.93**	**11825**	100%
PCJL-2, targets g^0	14.71	79940	100%	4.49	76950	100%	32.10	159913	100%	2.03	14309	100%
PCJL, constant depth	13.22	65691	100%	3.76	49048	100%	32.43	170636	100%	1.99	12615	100%

Evaluation metrics. We report the solving time T in seconds, the number L of LP iterations, and the percentage O of runs that prove optimality within the time limit. A run that exceeds the time limit is recorded as 300 seconds, and its LP iterations are counted up to termination. T and L are geometric means over the forty instances of a family and the three seeds. Percentage changes in T are paired over instances and reported with 95% bootstrap confidence intervals (Appendix D).

5.2 Comparison with Existing Methods

Table 1 summarizes the results on D. PCJL achieves the shortest solving time on all four families and proves optimality on every run. Compared with SCIP, it reduces the solving time by 35.7% on FCMCNF, 63.1% on MaxSAT, 3.3% on GISP, and 14.6% on CA, and the 95% intervals exclude zero on all families except GISP (Figure 3). None of the learned baselines is faster than SCIP on any family. GNN-Branch and GNN-Joint require fewer LP iterations than SCIP yet solve more slowly. PCJL reduces the number of LP iterations by 38.8% to 71.7% and translates this reduction into shorter solving times, because its learned components account for only 2.3% to 10.9% of the solving time (Table 9). Appendix D reports the behavior of each baseline per family.

5.3 Cumulative Component Analysis

To evaluate the contribution of each part of PCJL, we introduce the changes one at a time (Table 2 and Figure 4). PCJL-ref deploys the reference policy π^0 (Appendix B) and adds no learned component. PCJL-B replaces the reference branching policy by the learned branching network of Section 4.2, PCJL-2 adds the node network, and the full model PCJL adds the depth policy of Section 4.3. The two controls below the rule respectively train the node network of PCJL-2 on the reference targets g^0 instead of g_F and replace the learned depth of PCJL by the constant depth that performs best on the training pool ($d = 6$ on all four families). Both controls use the branching checkpoint, the seed selection rule, and the threshold calibration of PCJL-2.

Table 2 shows that each cumulative step reduces the solving time on at least three families. The reference policy reduces the solving time of SCIP by 4.9% to 19.6% on FCMCNF, MaxSAT, and CA. On GISP, it requires 3.5% more solving time with almost the same number of LP iterations. This increase is consistent with the per-node overhead of its Python node selector. Every later row

Table 3: Performance on the larger pool E, without retraining or recalibration on E. Metrics are as in Table 1. The smallest recorded T and L of each family are in bold and are interpreted jointly with O.

	FCMCNF			MaxSAT			GISP			CA		
Policy	T	L	O	T	L	O	T	L	O	T	L	O
SCIP	69.43	355725	91%	19.27	296121	100%	55.50	413047	100%	**43.88**	461379	98%
GNN-Node	67.34	346138	92%	20.34	302059	100%	**55.40**	401394	100%	45.13	488814	98%
GNN-Branch	64.15	**185210**	92%	51.57	336360	98%	167.34	**207030**	85%	207.16	**262208**	29%
GNN-Joint	108.96	256476	71%	85.80	247871	99%	247.90	235041	53%	202.43	426754	33%
PCJL	**49.79**	197072	88%	**11.13**	**142367**	100%	55.81	262622	99%	52.44	328069	93%

uses the same selector. The learned branching network reduces the solving time of PCJL-ref by 6.9% to 16.9% and its LP iterations by 10.4% to 46.0% on all four families. The node network reduces the solving time further by 9.9% on FCMCNF, 40.6% on MaxSAT, and 1.0% on CA. On GISP, it increases the LP iterations by 1.6% and the solving time by 3.9%. The depth policy reduces the solving time of PCJL-2 by 1.5% to 11.7% and is faster than the best constant depth on every family. The first control shows the role of the node targets. The node network fitted to the policy-conditioned targets requires 18.0%, 26.9%, and 16.8% less LP work than the network fitted to the reference targets on FCMCNF, MaxSAT, and CA. On GISP, the two targets differ by 0.6%. On FCMCNF and CA, the network fitted to the reference targets even requires 3.3% and 18.9% more LP work than PCJL-B and thus performs worse than no node network. Appendix D reports further comparisons.

5.4 Generalization to Larger Instances

We further evaluate the trained policies on the larger instances in E without retraining or recalibration (Table 3). The number of variables grows by a factor of about 1.2 to 2.0 (Table 7). PCJL records the smallest capped solving time on FCMCNF and MaxSAT, 28.3% and 42.2% below SCIP. On FCMCNF, however, it proves optimality on 88% of the runs, compared with 91% for SCIP and 92% for GNN-Branch. On GISP, its solving time is within 0.6% of SCIP, and the interval includes zero. CA doubles the instance size. On this family, PCJL is 19.5% slower than SCIP and proves optimality on 93% of the runs compared with 98%. GNN-Branch and GNN-Joint degrade more severely and prove optimality on only 29% and 33% of the CA runs. Appendix D discusses the CA result.

6 Conclusion

This paper presents PCJL, a method that learns node selection, variable branching, and the depth of each descent in branch-and-bound from measured continuation costs. Because the cost of a node is defined only for a given branching decision at that node, PCJL fits and freezes the branching network first and then measures node targets under the actions of this network. A common reference continuation makes all later decisions. A gradient-boosted regressor trained under the frozen pair then overrides the fixed plunging rule whenever it predicts a cheaper depth. All learned components act inside SCIP, and its optimality certificates remain valid. PCJL is the fastest of the compared methods on all four benchmark families and reduces the solving time of SCIP by 15% to 63% on three of them. Each cumulative configuration change reduces the solving time on at least three families, and the learned depth is faster than the best constant depth on all four. On FCMCNF, MaxSAT, and CA, the policy-conditioned targets require less LP work than the reference targets. On larger instances, PCJL keeps the smallest capped solving time on FCMCNF and MaxSAT, proves optimality less often than SCIP on FCMCNF, and becomes slower on CA. Several limitations remain. On GISP, the Python implementation of the reference node selector adds overhead, and the interval of the relative time change includes zero. The node targets are measured on sampled states under a fixed reference continuation, and the prediction for the reference candidate is not fitted. The bounds of Appendix A concern a single decision. The results come from one selected model per family, and the comparison with the learned baselines does not equalize training cost.

AI use statement

In this work, we used generative AI tools for language editing of the manuscript. We have not used generative AI tools for research ideation, experiment design, code generation, or data analysis.

We have reviewed all AI-assisted text, and we take responsibility for the final content of this work, including text, claims, and artifacts.

REFERENCES

A. M. Alvarez, Q. Louveaux, and L. Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

C. Ansótegui and J. Gabàs. WPM3: An (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017.

K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, and et al. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, 2021.

A. Chmiela, E. B. Khalil, A. Gleixner, A. Lodi, and S. Pokutta. Learning to schedule heuristics in branch and bound. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

M. Colombi, R. Mansini, and M. Savelsbergh. The generalized independent set problem: Polyhedral analysis and solution approaches. *European Journal of Operational Research*, 260(1):41–55, 2017.

S. Du, J. Tong, and W. Fan. Learning efficient branch-and-bound for solving mixed integer linear programs. *Applied Soft Computing*, 172:112863, 2025.

C. A. Floudas and X. Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139:131–162, 2005.

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

P. Gupta, M. Gasse, E. B. Khalil, M. P. Kumar, A. Lodi, and Y. Bengio. Hybrid models for learning to branch. In *Advances in Neural Information Processing Systems*, 2020.

C. Hansknecht, I. Joormann, and S. Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

H. He, H. Daumé III, and J. M. Eisner. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing Systems*, 2014.

M. Hewitt, G. L. Nemhauser, and M. W. P. Savelsbergh. Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. *INFORMS Journal on Computing*, 22(2): 314–325, 2010.

E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, pp. 724–731, 2016.

Y. Kuang, J. Wang, H. Liu, F. Zhu, X. Li, J. Zeng, J. Hao, B. Li, and F. Wu. Rethinking branching on exact combinatorial optimization solver: The first deep symbolic discovery framework. In *International Conference on Learning Representations*, 2024.

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

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pp. 66–76, 2000.

S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano. PySCIPOpt: Mathematical programming in Python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, Lecture Notes in Computer Science, pp. 301–307. Springer, 2016.

A. Mattick and C. Mutschler. Reinforcement learning for node selection in branch-and-bound. *Transactions on Machine Learning Research*, 2024.

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

C. W. F. Parsonson, A. Laterre, and T. D. Barrett. Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 4061–4069, 2023.

A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms Workshop at NeurIPS 2020*, 2011.

L. Scavuzzo, F. Y. Chen, D. Chételat, M. Gasse, A. Lodi, N. Yorke-Smith, and K. Aardal. Learning to branch with tree MDPs. In *Advances in Neural Information Processing Systems*, 2022.

T. Schouwenaars, B. De Moor, E. Feron, and J. How. Mixed integer programming for multi-vehicle path planning. In *2001 European Control Conference*, pp. 2603–2608, 2001.

J. Song, R. Lanka, A. Zhao, A. Bhatnagar, Y. Yue, and M. Ono. Learning to search via retrospective imitation. *arXiv preprint arXiv:1804.00846*, 2018.

T. Wu, K. Akartunalı, J. Song, and L. Shi. Mixed integer programming in production planning with backlogging and setup carryover: Modeling and algorithms. *Discrete Event Dynamic Systems*, 23: 211–239, 2013.

K. Yilmaz and N. Yorke-Smith. A study of learning search approximation in mixed integer branch and bound: Node selection in SCIP. *AI*, 2(2):150–178, 2021.

G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

S. Zhang, S. Zeng, S. Li, F. Wu, and X.-Y. Li. Learning to select nodes in branch and bound with sufficient tree representation. In *International Conference on Learning Representations*, 2025.

A Theoretical Analysis

This appendix states the two results cited in Sections 3.3 and 4.2. Both concern one decision under a fixed continuation, not a complete solve.

A.1 Node Ordering under Different Branching Maps

This part formalizes the claim of Section 3.3 that the cost of a node depends on the branching map. Fix a state S, a continuation ρ, and completed candidate costs $q_\rho(S, n, v)$, and write $g_\beta(S, n) = q_\rho(S, n, \beta(S, n))$ for a branching map β. The targets g^0 and g_F of Section 4.2 are g_β with $\beta = \beta^0$ and $\beta = \hat{\beta}_F$.

Proposition 1 (Conditional reversal of node ordering)**.** *Suppose two open nodes n and n' admit candidate costs a_1, a_2 and b_1, b_2, respectively, and $a_1 < b_1 \le b_2 < a_2$. If the class of branching maps can choose these candidate actions independently at the two nodes, there are maps β_1, β_2 such that*

$$g_{\beta_1}(S, n) < g_{\beta_1}(S, n'), \qquad g_{\beta_2}(S, n) > g_{\beta_2}(S, n'). \tag{18}$$

Proof. Let β_1 select the candidates with costs a_1 and b_2. Let β_2 select those with costs a_2 and b_1. The inequalities follow directly from the assumed ordering. □

Remark. The proposition is a sufficient condition rather than a construction of an MILP instance. It shows that the decision formulation does not fix a node ordering independently of the branching map. Learning node selection for a fixed branching map nevertheless remains well defined.

A.2 Prediction Accuracy at One Decision

This part bounds the one-step loss of the minimizers of Section 4.2 at a state S. Let $\overline{\mathcal{C}}_V(H_n) = \mathcal{C}_V(H_n)$ at a node that requires branching and $\overline{\mathcal{C}}_V(H_n) = \{\perp\}$ at a node that closes. The pairs available to the learned policies are

$$\mathcal{A}_J(S) = \{(n, v) : n \in \mathcal{C}_N(S),\, v \in \overline{\mathcal{C}}_V(H_n)\}.$$

Let $\hat{\beta}_F$ be the branching map of Equation (13), $\widetilde{\beta}_F$ its unthresholded minimizer of Equation (12), and $\hat{n}$ the minimizer of Equation (14). At a closed node, $\hat{\beta}_F(S, n) = \perp$ and $\hat{b}(S, n, \perp)$ is set to $q_\rho(S, n, \perp)$ for notational purposes. This convention does not require evaluating the branching network at closed nodes.

Assumption 1 (Uniform prediction errors)**.** All costs in the following comparisons are defined, and

$$\max_{(n,v)\in\mathcal{A}_J(S)} |\hat{b}(S, n, v) - q_\rho(S, n, v)| \le \varepsilon_V, \qquad \max_{n\in\mathcal{C}_N(S)} |\hat{g}(S, n) - g_F(S, n)| \le \varepsilon_N.$$

Proposition 2 (One-step approximation bounds)**.** *Under Assumption 1,*

$$g_F(S, \hat{n}) \le \min_{n\in\mathcal{C}_N(S)} g_F(S, n) + 2\varepsilon_N \tag{19}$$

and

$$q_\rho(S, \hat{n}, \hat{\beta}_F(S, \hat{n})) \le \min_{(n,v)\in\mathcal{A}_J(S)} q_\rho(S, n, v) + 2\varepsilon_N + 2\varepsilon_V + \gamma_V. \tag{20}$$

Proof. For every $n \in \mathcal{C}_N(S)$, Assumption 1 and the minimization of $\hat{g}$ give

$$g_F(S, \hat{n}) \le \hat{g}(S, \hat{n}) + \varepsilon_N \le \hat{g}(S, n) + \varepsilon_N \le g_F(S, n) + 2\varepsilon_N.$$

This proves Equation (19). Let $v^\star(n)$ minimize the true cost at n over $\overline{\mathcal{C}}_V(H_n)$. If $\hat{\beta}_F(S, n) = \widetilde{\beta}_F(S, n)$, the same argument applied to $\hat{b}$ gives

$$g_F(S, n) \le \hat{b}(S, n, \widetilde{\beta}_F(S, n)) + \varepsilon_V \le \hat{b}(S, n, v^\star(n)) + \varepsilon_V \le q_\rho(S, n, v^\star(n)) + 2\varepsilon_V.$$

If instead $\hat{\beta}_F(S, n) = \beta^0(S, n)$, the threshold in Equation (13) gives $\hat{b}(S, n, \beta^0(S, n)) < \hat{b}(S, n, \widetilde{\beta}_F(S, n)) + \gamma_V$, and therefore

$$g_F(S, n) \le \hat{b}(S, n, \beta^0(S, n)) + \varepsilon_V < \hat{b}(S, n, v^\star(n)) + \varepsilon_V + \gamma_V \le q_\rho(S, n, v^\star(n)) + 2\varepsilon_V + \gamma_V.$$

In both cases, $g_F(S, n) \le q_\rho(S, n, v^\star(n)) + 2\varepsilon_V + \gamma_V$. Since $q_\rho(S, \hat{n}, \hat{\beta}_F(S, \hat{n})) = g_F(S, \hat{n})$, the two inequalities together prove Equation (20). □

Remark. The bound concerns the learned pair before the node threshold is applied. It shows that the one-step loss is controlled by the two prediction errors and the branching threshold, and that the node error is measured against targets obtained under the deployed branching actions and the reference continuation.

Limits of the bounds. The bounds concern costs under a fixed continuation at a single state. They do not accumulate into a bound on total work, do not cover censored costs, and do not imply a reduction in solving time. Policy-conditioned targets determine which branching action a node target evaluates, but they do not remove prediction error, shifts in the visited-state distribution, or the gap between the replay continuation and repeated deployment. The node threshold γ_N and the depth policy are not covered by these results.

B Implementation and Measurement Protocol

This appendix specifies the measurement and fitting procedures of Section 4.

B.1 Replay Protocol and Data Splits

This part specifies the replays behind Definition 1 and Equation (17). In SCIP, plunging is part of the node selector. A custom node selector therefore removes the native plunging rule (Bestuzheva et al., 2021). GNN-Node and GNN-Joint are also implemented as custom node selectors and thus replace this rule as well. The reference policy reimplements this rule inside a node selector that orders the open nodes by estimate. The reimplementation uses three parameters of the native rule, namely the maximal plunging depth, the maximal plunging quotient, and the frequency of best-node selections. During a descent, the selector takes the child of the last processed node with the best estimate, or the sibling with the best estimate when no child is open. Because the quotient test compares bounds in the original objective sense, it takes effect only on the minimization family FCMCNF. On the three maximization families, the quotient test never ends a descent, and the reimplemented rule therefore differs from the native rule. The reference node selector is implemented in Python. The native selector of SCIP is implemented in C.

A branching replay forces the pair (n, v) at a recorded state S. A node replay forces n and applies $\hat{\beta}_F(S, n)$ from the frozen branching network. At a node that is closed during processing, $\hat{\beta}_F(S, n) = \perp$. Node replays are needed only where $\hat{\beta}_F(S, n) \neq \beta^0(S, n)$, because g^0 is reused elsewhere (Section 4.2).

Every replay returns to its continuation and shares one no-intervention baseline per state. It counts only the work performed after the restored prefix, including the processing of n. Restoring the prefix reproduces the solver state that subsequent decisions require, not only the MILP subproblem. Replays use a single solver thread and fixed seeds. A record is kept only when the numbers of processed nodes and LP iterations at S match the recorded ones.

A depth replay makes the requested d selections with the rule above and then returns control to the frozen pair (Section 4.3). Because depth costs use the frozen pair rather than π^0 as their continuation, they are not compared numerically with node or branching costs.

The node limit of each replay is the larger of $1{,}000$ and 20 times the nodes processed by the reference solve. Each replay is also limited to 300 seconds of wall-clock time. A run that ends before the solver proves optimality or infeasibility is censored. A record enters Definition 1 only when both of its runs complete. The added one in Equation (8) accommodates runs with zero LP iterations. Dividing by the no-intervention run makes costs comparable across states of different size.

Table 6 reports the retained counts per stage. No replay was censored for the reported models.

B.2 Features, Losses, and Calibration

Table 4 lists the descriptors of Sections 4.1 and 4.3 by their availability. Root features and root embeddings are cached. Node, candidate, and depth features are assembled when their decisions occur. Cut rows added after the cache is constructed are not included in the cached graph. The node descriptor consists of seventeen frontier quantities and ten quantities of the candidate node. Besides eighteen quantities of the search state, the depth regressor receives the candidate depth as $\log_2(d+1)$.

A scalar t measured relative to the best node estimate b_{est} is normalized as $(t - b_{\text{est}})/\max(|b_{\text{est}}|, 1)$. Counts, gaps, and solution ages enter as $\log(1 + \max(x, 0))$. Non-finite values and values of magnitude at least 10^{18} are set to zero.

Losses, initialization, and calibration. Both networks use the same fitting loss. For the measured costs $y_1, \ldots, y_m$ of the actions in one record and their predictions $\hat{y}_1, \ldots, \hat{y}_m$, it is

$$\mathcal{L}_{\text{fit}} = \text{mean}_j \, H_\delta(\hat{y}_j - y_j) + \eta \, \text{mean}_{(j,k):\, y_j + \tau < y_k} \, \text{softplus}(\hat{y}_j - \hat{y}_k), \tag{21}$$

where H_δ is the Huber function. The first term fits the costs, and the second penalizes predictions that reverse a clear difference in cost within the record. A branching record consists of the candidates in $\mathcal{C}_V(H_n)$ at one node. A node record consists of the nodes in $\mathcal{C}_N(S)$ at one state. The branching network minimizes $\mathcal{L}_{\text{fit}}$ with $y = q_\rho(S, n, v)$ on records that exclude the reference candidate, because that candidate has zero cost at the reference node by construction. Its prediction $\hat{b}(S, n, \beta^0(S, n))$ is nevertheless produced by the same network and enters the threshold of Equation (13) and the consistency term. Assumption 1 treats this prediction like any other, and its calibration is not verified

Table 4: Feature groups and their availability.

Group	Features and normalization
Root variables	Variable type; objective coefficient divided by objective norm; degree divided by maximum variable degree.
Root constraints	Row–objective cosine; right-hand side minus row constant divided by row norm; normalized constraint degree.
Root edges	Nonzero coefficient divided by row norm.
Root cache	Frozen variable embedding e_v.
Frontier, r_N	Frontier size; child/sibling/leaf proportions; estimate spread; mean and maximum depth; focus depth; work since incumbent improvement; primal–dual gap; solution count; processed-node and LP-work counters. Log and relative variants are used.
Node, r_N	Estimate and local dual bound relative to the best estimate; their difference; node depth and depth relative to the focus; child/sibling/leaf indicators; reference rank and top-rank indicator.
Reference rank, r_V	Rank of the candidate under the reference branching network.
LP candidate, r_V	LP value; fractionality; finite-bound and at-bound indicators; basis status; reduced cost divided by objective norm; bounds with magnitude at least 10^{19} count as infinite, and the at-bound tolerance is 10^{-9}.
Local structure, r_V	Fractional-neighbor counts and fractional mass over incident constraints; their candidate-count-normalized variants; variable degree.
Depth, r_D	Depth of the node that would start the descent, maximum tree depth, and their ratio; numbers of open children, siblings, and leaves, and the log of their total; incumbent indicator and relative gap; bound of that node relative to the best bound, scaled by the gap and by the objective scale; its estimate relative to the best estimate; maximal plunging depth of the reference rule; logs of the processed nodes, LP iterations, and solutions; position in the best-node cycle; log of the number of selection calls.

separately. With $y = g_F(S, n)$, the node network minimizes

$$\mathcal{L}_N(f_N) = \mathcal{L}_{\text{fit}} + \lambda_C \,\text{mean}\big(\hat{g}(S,n) - \hat{b}(S,n,\hat{\beta}_F(S,n))\big)^2, \tag{22}$$

where the branching network stays frozen and the second mean runs over the sampled nodes that require branching.

Because the final scalar layers of f_V and f_N are initialized to zero, all candidates receive equal predictions before training. Because ties are resolved by the reference ranking and the thresholds γ_N and γ_V are nonnegative, the untrained pair reproduces the reference actions. Reference rank enters as an input rather than as an additive score.

The node threshold is $\gamma_N = \max\{0,\, Q_{0.8}(\Delta_N)\}$, where $\Delta_N(S) = \hat{g}(S, n^0) - \hat{g}(S, \hat{n})$ ranges over the validation states and $Q_{0.8}$ denotes the empirical 80% quantile. The node policy therefore departs from n^0 on at most about 20% of the validation states. The branching threshold is $\gamma_V = \max\{0,\, Q_{0.8}(\Delta_V)\}$ with $\Delta_V(S,n) = \hat{b}(S,n,\beta^0(S,n)) - \hat{b}(S,n,\widetilde{\beta}_F(S,n))$ over the branching decisions of the same validation states. It is calibrated after the branching network has been frozen and before the node targets are measured. Table 5 lists both thresholds. Of three training seeds per family, PCJL keeps the one whose pair incurs the least LP work on the validation pool.

B.3 Algorithms and Configuration

This part gives the fitting and deployment procedures of Section 4.

Algorithm 1 applies the measurement of Definition 1 in the order that keeps node targets matched to the branching policy.

Algorithm 2 gives one deployed selection. Reference rankings resolve node and branching ties, and depth ties are resolved toward the smaller depth. PCJL-2 in Section 5.3 runs this algorithm without the depth policy and keeps the reference plunging rule for its descents.

Configuration. Table 5 lists the configuration of PCJL, and Table 6 reports the replay and fitting volumes behind the fitted models.

Algorithm 1 Staged fitting of PCJL

Require: frozen graph encoder and reference policy; a training pool and a held-out validation pool
Ensure: frozen θ_B, γ_V, f_N, γ_N, and f_D

Collect branching costs under π^0 by matched replay, Definition 1
Fit $\theta_B = (E_N, P, E_V, f_V)$ with Equation (21); freeze θ_B
Calibrate γ_V on the validation pool; freeze the branching map $\hat{\beta}_F$ of Equation (13)
for recorded state S in the training pool **do**
    for sampled $n \in \mathcal{C}_N(S)$ **do**
        Determine $\hat{\beta}_F(S, n)$ with the frozen branching map, Equation (13)
        if $\hat{\beta}_F(S, n) = \beta^0(S, n)$ **then**
            Set $g_F(S, n) = g^0(S, n)$
        else
            Replay n with $\hat{\beta}_F(S, n)$ under π^0
            Store $g_F(S, n)$ only if both runs complete
        end if
    end for
end for
Fit f_N with Equation (22); calibrate γ_N; freeze the node–branching pair
Collect depth costs under the frozen pair, Equation (17)
Fit f_D by squared error

Algorithm 2 One deployed PCJL selection

Require: state S; frozen networks, f_D, γ_N, and γ_V; descent type and counter c
Ensure: the node processed at this step and, if it requires branching, its branching variable

if no descent is active and a child or sibling of the last processed node is open **then**
    Choose $\hat{a}$ by Equation (16)
    if $\hat{a} \neq \varnothing$ **then**
        Start a learned descent with $c \leftarrow \hat{a}$
    else
        Start a reference descent
    end if
end if
if a learned descent is active, $c > 0$, and a child or sibling of the last processed node is open **then**
    Select the child with the best estimate, or else the sibling with the best estimate
    Set $c \leftarrow c - 1$
else if a reference descent is active and the reference plunging rule continues it **then**
    Select the node that the reference plunging rule chooses
else
    End any active descent; select n^0 if S is not contested; otherwise select $\hat{n}$ by Equation (14), or n^0 if $\hat{g}(S, n^0) - \hat{g}(S, \hat{n}) \leq \gamma_N$
end if
Compute h_n; process n to obtain H_n
if n is closed during processing **then**
    return
end if
if the root cache has not been constructed **then**
    Construct the cache at this first branching call
end if
Determine $\hat{\beta}_F(S, n)$ by Equation (13); branch on $\hat{\beta}_F(S, n)$

Table 5: Configuration of PCJL.

Component	Setting
Root encoder	Reference branching network with width 64 and one message-passing round for FCMCNF, MaxSAT, and GISP, and width 128 and three rounds for CA. Its frozen root embeddings serve as e_v, so d_e is 64 or 128.
Reference training	Soft-target temperature 0.2; Adam, learning rate 10^{-3}, 40 epochs.
Branching network	E_N: two-layer multilayer perceptron of width 64 on 27 node features. E_V: two-layer multilayer perceptron of width 64 on e_v, 11 LP features, 5 local-structure features, and the reference rank, with layer normalization of the last 17 inputs. P: $64 \to 64$ without bias. f_V: multilayer perceptron of width 64 on $[z; u; z \odot u]$.
Node network	f_N: three linear layers of width 64 with a zero-initialized output layer.
Optimization	Adam with learning rate 10^{-3} in both stages; gradients accumulated over 16 states; at most 200 epochs; early stopping on the validation loss with patience 25.
Losses	$\delta = 0.05$, $\tau = 0.01$, and $\eta = 1$ in Equation (21); $\lambda_C = 1$ in Equation (22).
Continuation	The reference policy π^0 for node and branching costs and the frozen pair for depth costs.
Sampling and candidate sets	Up to eight contested states per instance, evenly spaced along the reference trajectory. In a contested state, $\mathcal{C}_N(S)$ contains the five best open nodes by estimate; otherwise it contains only n^0. $\mathcal{C}_V(H_n)$ contains the five best candidates by reference rank.
Contested states	The estimates of the two best open nodes differ by at most 10^{-3} in relative terms. The same threshold is used in collection and at deployment.
Node threshold	$\gamma_N = \max\{0, Q_{0.8}(\Delta_N)\}$ on the validation states, so that the node policy departs from n^0 on at most about 20% of them; 0, 0.06218, 0.02221, and 0.00425 for FCMCNF, MaxSAT, GISP, and CA.
Branching threshold	$\gamma_V = \max\{0, Q_{0.8}(\Delta_V)\}$ on the validation decisions after θ_B is frozen, so that the branching policy departs from β^0 on at most about 20% of them; 0.12355, 0.01236, 0, and 0.26322 for FCMCNF, MaxSAT, GISP, and CA.
Depth actions	$\mathcal{D} = \{0, 2, 6, 15, 1000\}$, where 1000 exceeds every tree depth observed in the pools and thus follows the path to its end. The no-intervention action $\varnothing$ has predicted cost 0 and keeps the reference plunging rule.
Constant depth (control)	$d = 6$ on all four families.
Depth collection	Up to sixty training instances per family and six evenly spaced descent starts per instance; 46, 59, 60, and 58 instances contribute descent starts in FCMCNF, MaxSAT, GISP, and CA.
Depth regressor	One histogram-based gradient-boosted regressor per family on the eighteen state quantities and $\log_2(d+1)$; squared loss; 200 boosting iterations; learning rate 0.05; maximum depth 4; minimum leaf samples 20; $L_2 = 1$.
Replay budget	The larger of 1,000 nodes and 20 times the nodes of the reference solve; 300 seconds.

Table 6: Replay and fitting volumes per family. Branching and node records count the training states and, in parentheses, the validation states. Depth groups count the training descent starts. Epochs are the selected epochs.

Family	Branch records	Node records	Depth groups	Epochs (branch / node)
FCMCNF	419 (150)	377 (148)	252	7 / 10
MaxSAT	450 (150)	428 (137)	354	16 / 17
GISP	750 (300)	679 (267)	358	4 / 6
CA	728 (240)	660 (221)	333	7 / 9

C Benchmarks and Instance Pools

This appendix describes the benchmark families and the instance pools of Section 5.1.

Benchmark families. All four families are generated synthetically. FCMCNF, MaxSAT, and GISP instances are produced with the generation code of Labassi et al. (2022) (commit `da802e8d`), and CA instances with the generator of Ecole 0.8 (Prouvost et al., 2011). For MaxSAT and CA, the training and validation pools use the settings of D. For FCMCNF and GISP, they contain smaller instances of 17 nodes and of 60 to 70 vertices, respectively. The pool E uses a larger setting of the same generator.

FCMCNF. The fixed-charge multicommodity network flow problem (Hewitt et al., 2010) routes a set of commodities through a capacitated network at minimum cost, where each arc that carries flow incurs a fixed cost in addition to a cost proportional to the flow. The networks are directed Erdős–Rényi graphs with edge probability 0.33, and the number of commodities lies between $n_{\max}$ and $1.5\,n_{\max}$ for a largest node count $n_{\max}$. Costs are drawn from $[11, 50]$, demands are drawn from $[10, 100]$, and the fixed-cost ratio is 100. Pool D has 20 to 22 nodes and 22 to 33 commodities, and pool E has 24 to 25 nodes and 25 to 37 commodities.

MaxSAT. The weighted partial maximum satisfiability problem (Ansótegui & Gabàs, 2017) seeks a truth assignment that satisfies all hard clauses and maximizes the total weight of the satisfied soft clauses. The instances encode maximum-cut problems on Erdős–Rényi graphs with edge probability 0.6 and with 70 to 78 nodes in D or 85 to 95 nodes in E.

GISP. The generalized independent set problem (Colombi et al., 2017) selects vertices of a graph to maximize their total revenue minus the removal costs of edges. No permanent edge may join two selected vertices. A removable edge may do so only if its removal cost is paid. The instances are built on Erdős–Rényi graphs with edge probability 0.6 under the revenue and cost scheme SET2 with parameter 100, and a fraction $\alpha = 0.5$ of the edges is removable. Pool D has 100 to 105 vertices, and pool E has 108 to 112.

CA. The combinatorial auction problem (Leyton-Brown et al., 2000) selects a set of bids, no two of which request the same item, that maximizes the total price of the accepted bids. Pool D has 100 items and 500 bids, and pool E has 200 items and 1000 bids.

Table 7 reports the average dimensions of the resulting instances.

Instance pools. Table 8 reports the instances used at each stage. Branching and node records are collected on the same training instances (C_{tr}), and depth records on a subset of them (C_{dp}). The validation states (C_{val}) come from a separate validation pool. A comparison of instance contents finds no evaluation instance among the training, validation, or encoder-pretraining instances. The forty evaluation instances of each family are obtained by sorting the generated pool by file name and taking forty evenly spaced instances. The generated pools contain 103, 200, 130, and 100 instances for D and 62, 50, 45, and 50 for E, in the order FCMCNF, MaxSAT, GISP, and CA.

Only four CA instances in D are solved by SCIP at the root node. These instances never invoke the branching rule and therefore never build the root cache.

Table 7: Average dimensions of the forty evaluation instances of each pool, before presolving.

	D		E		E/D	
Family	Variables	Constraints	Variables	Constraints	Var.	Con.
FCMCNF	3933	714	6052	945	1.5	1.3
MaxSAT	1823	1749	2614	2525	1.4	1.4
GISP	1648	3109	1896	3577	1.2	1.2
CA	500	192	1000	385	2.0	2.0

Table 8: Instances per family that contribute branching and node records (C_{tr}), depth records (C_{dp}), and validation instances (C_{val}), and evaluated instances in D and E.

Family	C_{tr}	C_{dp}	C_{val}	D	E
FCMCNF	120	46	50	40	40
MaxSAT	90	59	30	40	40
GISP	150	60	60	40	40
CA	150	58	50	40	40

D Evaluation Protocol and Additional Results

This appendix specifies the paired comparisons of Section 5 and reports the inference cost of PCJL.

Machine and load. Each test instance is solved with the solver seeds 9, 17, and 42 under a time limit of 300 seconds. All evaluations run on an Intel Xeon Platinum 8375C processor (2.90 GHz) in a container limited to 12 cores and 62 GB of memory. Twelve single-threaded solves run in parallel, each pinned to its own core, and all learned components run on the CPU. The reported times therefore correspond to a fully loaded machine rather than to an isolated solve. The configurations of one table are evaluated in the same interleaved batch and thus under the same load, and Tables 1 and 2 are evaluated in the same batch. Solving time is the quantity that matters in practice, and the number of LP iterations reflects the search effort independently of the overhead of model inference. For runs that reach the time limit, T and L are lower bounds on the true solving effort, and O indicates how often this truncation occurs. The value of an instance is the geometric mean over its three seeds, and the value of a family is the geometric mean over its forty instances.

Additional results. On D, GNN-Node is slower than SCIP on every family, by 2.1% on FCMCNF and by up to 21.7% on CA. GNN-Branch and GNN-Joint require 7.4% to 63.8% fewer LP iterations than SCIP, yet GNN-Branch is 2.7 to 3.0 times slower on three families and GNN-Joint is slower on all four. Du et al. (2025) make the same observation that a smaller search tree does not necessarily lead to a shorter solving time. Relative to the fastest learned baseline on each family, the reduction of PCJL ranges from 9.9% on GISP to 64.5% on MaxSAT. The smallest improvement is observed on GISP, where a 38.8% reduction in LP iterations yields a 3.3% reduction in solving time. This difference comes from the cost of the node selector on this family.

In the cumulative analysis of Table 2, PCJL-ref requires 28.6% to 45.0% fewer LP iterations than SCIP on FCMCNF, MaxSAT, and CA. On GISP, its LP work stays within 0.4% of SCIP, and its solving time is 3.5% longer. This overhead of the Python node selector persists in every subsequent row. The learned branching network reduces the solving time of PCJL-ref by 8.4% on FCMCNF, 16.9% on MaxSAT, 6.9% on GISP, and 7.9% on CA, and it lowers the number of LP iterations by 10.4% to 46.0% on every family. The node network lowers the number of LP iterations of PCJL-B by 15.3% on FCMCNF and 39.1% on MaxSAT. On GISP, the LP work rises by 1.6%. From PCJL-ref to PCJL-2, the LP work falls on all four families. In the target control, the solving time of PCJL-2 is 11.3%, 23.8%, 2.6%, and 3.4% shorter than that of the control on FCMCNF, MaxSAT, GISP, and CA. On FCMCNF, GISP, and CA, the control also requires more LP work than PCJL-B. It therefore performs worse than the configuration without a node network on these families. The two targets differ by $g_F(S,n) - g^0(S,n) = q_\rho(S,n,\hat{\beta}_F(S,n)) - q_\rho(S,n,\beta^0(S,n))$, the advantage of the deployed branching action at n. A node network fitted to g^0 ignores this advantage. The control

keeps the consistency term of PCJL-2, and this term still exposes its node network to part of the advantage. The measured differences are therefore more likely to understate than to overstate the effect of the target. The depth policy reduces the solving time of PCJL-2 by 3.3% on FCMCNF, 11.7% on MaxSAT, 3.4% on GISP, and 1.5% on CA. Its effect on the LP work is largest on MaxSAT and FCMCNF, where the number of LP iterations decreases by 19.9% and 11.0%. On GISP, the LP work stays within 0.2% of PCJL-2. Compared with the best constant depth ($d = 6$ on all four families), the learned depth is faster by 3.0% on CA and by up to 19.7% on MaxSAT, and it requires 6.3% to 11.2% fewer LP iterations.

On the larger CA instances, PCJL still requires 28.9% fewer LP iterations than SCIP. The additional runs that reach the time limit and are charged at 300 seconds may partly explain the increase in capped solving time. CA is the family with the largest size increase, and its thresholds were calibrated on instances half the size of those in E. On GISP, GNN-Joint proves optimality on only 53% of the runs. The low LP iteration counts of GNN-Branch and GNN-Joint on these families therefore reflect runs that terminate at the time limit rather than more efficient searches. GNN-Node stays within 6% of SCIP on every family but does not improve on it consistently.

Paired comparisons and uncertainty. Comparisons are paired by instance. For the solving times $t_{p,i,s} > 0$ of method p on instance i with solver seed s, let $\bar{t}_{p,i} = \exp(|\mathcal{S}|^{-1} \sum_{s \in \mathcal{S}} \log t_{p,i,s})$. The runtime ratio of p against a baseline b is

$$R_T(p, b) = \exp\left(\frac{1}{N} \sum_{i=1}^{N} \log \frac{\bar{t}_{p,i}}{\bar{t}_{b,i}} \right),$$

where $N = 40$ within a family, $\mathcal{S} = \{9, 17, 42\}$, and a ratio below one favors p. Because R_T equals the ratio of the family-level geometric means in Tables 1 and 3, the percentage changes in Section 5 are $100\,(R_T - 1)$. When runs reach the time limit, R_T compares times truncated at the limit rather than completion times.

Intervals are computed by a paired bootstrap over instances. Each of 10,000 resamples draws forty instances with replacement and keeps all seeds and all methods of a drawn instance together. Percentile 95% intervals are reported, and the bootstrap uses the seed 20260911. The intervals describe variation across test instances for fixed trained models. Because no reported cell contains a zero LP or node count, plain geometric means are used throughout.

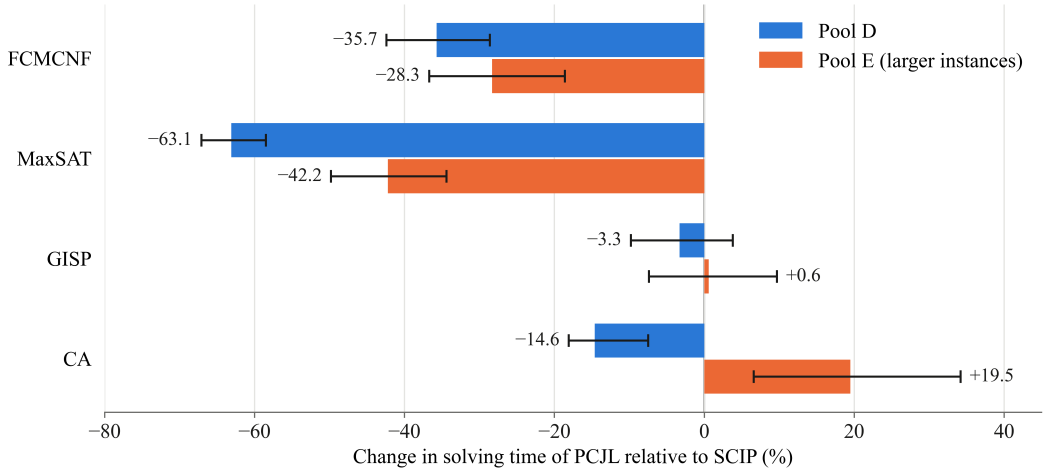

Figure 3: Change in the solving time T of Section 5.1 of PCJL relative to SCIP on the pools D and E. Bars show the paired estimates and whiskers the 95% bootstrap intervals (Appendix D). Negative values favor PCJL.

Inference cost. Table 9 reports the time that PCJL spends in its learned components on D, measured on the machine described above. The cached root representation is computed once at the root. The

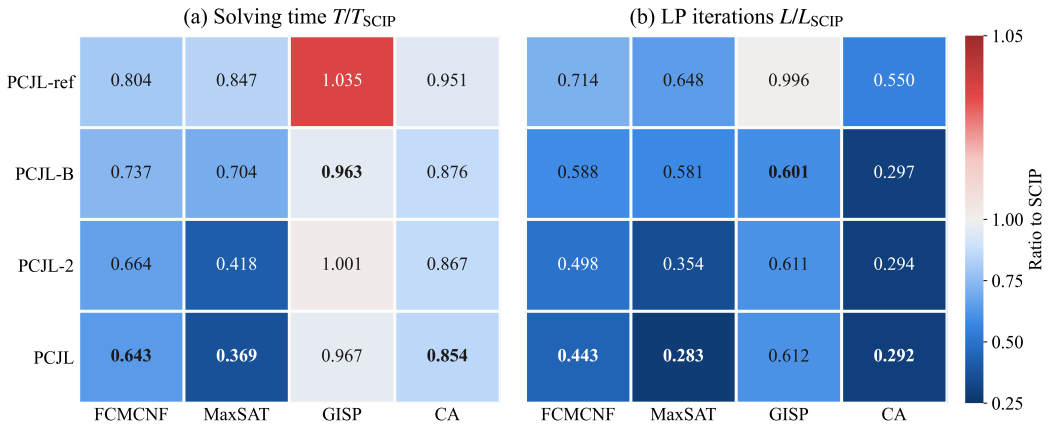

Figure 4: Cumulative rows of Table 2 on D. Each cell reports the ratio to SCIP of the solving time T (left) or the LP iterations L (right). Lower values are better, the shared color scale is centered at 1, and boldface marks the smallest ratio within each family and metric.

Table 9: Inference cost of the learned components of PCJL on D, measured inside the solves with twelve concurrent solves. The root cache is timed at its construction and the other components per call. The branching column gives the imitation-trained scorer called at every branching and the learned branching head. The last column reports the share of the solving time T spent in these components.

Family	Root cache (ms)	Node network (ms)	Branching network (ms, scorer / head)	Depth policy (ms)	Share (%)
FCMCNF	188.7	0.30	1.19 / 0.41	0.27	2.3
MaxSAT	96.4	0.28	2.29 / 0.38	0.25	10.9
GISP	139.3	0.29	0.89 / 0.41	0.28	5.5
CA	129.2	0.27	1.05 / 0.38	0.26	7.6

node network is evaluated only in contested states, the branching network at every node that requires branching, and the depth policy at the start of every descent. The share excludes the time spent in the reference node selector.