LEARNING TO SELECT PARTIAL ASSIGNMENTS FOR OPTIMIZATION MODULO THEORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Many applications built on satisfiability modulo theories (SMT) need optimal solutions under their existing logical constraints. Optimization Modulo Theories (OMT) provides this capability by optimizing numerical objectives over the formula's satisfying assignments. Its solvers must resolve logical alternatives while finding and establishing the best objective value permitted by the formula. We present NEUROMT, a framework for guiding OMT over linear integer arithmetic by selecting candidate partial assignments. Each candidate assigns truth values to a subset of arithmetic predicates, restricting the search while leaving numerical values open. We train a graph neural network to select among these candidates using the total solver cost of restricted optimization and subsequent exact search. The network scores each candidate from its assigned atoms, their truth values, and the formula representation. The selected assignment guides optimization to obtain an objective bound, followed by exact continuation under the original constraints. Experiments show reductions in total solver-stage time relative to direct Z3 on OMT benchmarks including knapsack and resource-constrained scheduling, with ratios from 11.4% to 94.4%.

1 INTRODUCTION

Satisfiability Modulo Theories (SMT) checks whether a logical formula admits an assignment consistent with background theories, supporting a wide range of applications such as program analysis, verification, software testing and etc. Many applications also require tight resource bounds or minimum-cost configurations [1, 2]. Optimization Modulo Theories (OMT) extends such capability to SMT by optimizing an objective over the formula's satisfying assignments [3].

We study OMT over linear integer arithmetic (LIA). A partial assignment specifies truth values for selected arithmetic predicates and narrows the search space. A predicate can constrain relations among several variables while leaving their individual values to the optimizer. For example, assigning true to a predicate $x + y \leq 5$ restricts the sum without fixing either variable; assigning false to the predicate imposes $x + y \geq 6$. Unassigned predicates remain governed by the original formula, allowing one restriction to retain several logical alternatives. This makes atom-level assignments a flexible way to guide search without constructing a complete numerical solution. Its value depends on the combined cost of optimizing the restricted problem and completing the search under the original constraints to certify global optimality.

Choosing an assignment requires deciding which predicates to include, how many to assign, and which truth values to use. These choices interact: atoms may share variables, occur in the same clause, or impose competing restrictions on the objective. With m selectable atoms, there are 3^m possible partial assignments before consistency is considered, since each atom can be true, false, or unassigned. Exhaustively measuring their costs would itself require many exact solves. Optimal solutions alone do not provide this ranking: two assignments compatible with the same optimum can induce different amounts of solver work.

The cost also depends on what happens after the restriction is removed. A restricted problem that is easy to solve may produce a weak objective bound, leaving substantial work for the remaining search. A more expensive initial optimization may instead produce a bound that eliminates much of that work. Training on the complete optimization cost captures this tradeoff and directs selection toward assignments that help finish the task.

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.

Existing methods use partial assignments in different ways. Masina and Sebastiani reduce satisfying assignments within an OMT solver to enlarge the regions passed to theory optimization [4]. Ashera predicts partial numerical assignments and alternates SMT feasibility search with integer-programming optimization of extracted conjunctive regions [5]. We instead select truth assignments to arithmetic predicates and optimize the restricted original formula, preserving the Boolean structure that the assignment leaves open.

NEUROMT makes this selection tractable by constructing a finite set of candidate partial assignments. A shared graph neural network (GNN) encodes the formula once and scores each candidate using its assigned atoms and truth values. A construction rule can yield different assignments across formulas, so the network evaluates each assignment in its formula context. This lets it distinguish candidates produced by the same rule. Training uses measured costs of restricted optimization and subsequent exact search. At inference, the network selects an assignment without solving every candidate. The restriction is temporary: after using it to obtain a bound, the optimizer resumes under the original constraints. The selected assignment can therefore help the search even if it excludes every global optimum.

Our contributions are threefold. First, we propose a hybrid framework for OMT guidance that combines learned partial-assignment selection with exact optimization. Second, we design structural rules to generate candidate assignments and a GNN selector trained on the combined cost of restricted optimization and subsequent global search. Third, we implement the framework and evaluate it on OMT benchmarks including knapsack and three scheduling benchmark classes, achieving solver-stage PAR-2 reductions of 11.4% on knapsack and up to 94.4% on scheduling relative to direct Z3.

2 BACKGROUND AND RELATED WORK

2.1 OMT AND PARTIAL ASSIGNMENTS

Let $X = \{x_1, \ldots, x_n\}$ be integer variables. We use quantifier-free linear integer arithmetic (LIA), with terms t and formulas φ given by

$$t ::= c \mid x_i \mid t + t \mid c\, t, \qquad c \in \mathbb{Z},\ x_i \in X,$$
$$\varphi ::= t \le t \mid t = t \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi.$$

An assignment $x \in \mathbb{Z}^n$ evaluates terms by integer arithmetic and formulas by the usual Boolean semantics; $x \models F$ means that F is true under x.

An OMT instance $I = (F, obj)$ consists of such a formula F and a linear objective obj over X. We use maximization; minimization is converted by negating the objective. For feasible instances, we consider finite, attained optima:

$$opt = \max\{obj(x) : x \models F\}. \tag{1}$$

SMT combines propositional search and theory reasoning to find a model of F [6, 7]. OMT finds a satisfying assignment with the best objective value. The Boolean abstraction treats each arithmetic atom as a propositional variable, but its truth values must remain consistent with integer arithmetic. For example, $x \le 0$ and $x \ge 1$ cannot both be true. Theory reasoning detects such conflicts, while the logical formula determines which combinations of atoms need to hold.

For generation and graph encoding, arithmetic comparisons are normalized to signed inequality atoms $a_j : \boldsymbol{w}_j^\top \boldsymbol{x} \le b_j$; equalities split into two inequalities. Clauses C represent their Boolean structure. Exact optimization uses the original formula F. Let A denote the atom set and $U \subseteq A$ the atoms not fixed by root propagation. This preprocessing derives atom truth values forced by the formula through Boolean implications and integer bounds; the atoms in U remain available for candidate construction. A partial assignment is

$$\mathcal{A} = \{(a, b_a) : a \in A^\star\}, \qquad A^\star \subseteq U, \quad b_a \in \{\top, \bot\}. \tag{2}$$

We also write $\mathcal{A}$ for the conjunction of these signed literals, often called a cube. Assigning false negates the corresponding predicate; the empty assignment leaves F unchanged. For example, assigning true to $x + y \le 5$ restricts the sum while leaving both numerical values open; assigning

false imposes $x + y \geq 6$ over integers. A candidate can therefore constrain several variables without fixing any of them. It need not assign enough atoms to satisfy every clause: the remaining clauses and unassigned atoms are still part of $F \wedge \mathcal{A}$ and are handled by the optimizer.

If $F \wedge \mathcal{A}$ is feasible, its optimum satisfies

$$opt^{\star} = \max\{obj(x) : x \models F \wedge \mathcal{A}\} \leq opt. \tag{3}$$

A partial assignment may exclude all global optima, and arithmetic constraints can make it infeasible. Its usefulness depends on the cost of optimizing this restriction and completing the search under F. Feasibility and preservation of a global optimum are separate properties: a restriction may admit solutions but retain only suboptimal ones. An optimum-preserving restriction can also be computationally expensive, because finding its best solution and establishing optimality still require search. These distinctions let us assess candidate quality by the solver work it induces, while leaving correctness to exact optimization.

2.2 Related Work

Partial assignments in OMT. The reduced assignments of Masina and Sebastiani still propositionally satisfy the formula [4]. They are constructed inside the solver to increase the region available to theory optimization. Our candidates are additional restrictions chosen before optimization; unassigned atoms retain their role in the original Boolean formula.

Neural optimization guidance. Ashera learns partial numerical assignments from optimal solutions. Its neural diver uses SMT to complete these assignments. Its optimization engine alternates SMT search under an improving bound with Gurobi optimization of conjunctive regions extracted from satisfying models [5]. NeurOMT learns to select candidate partial assignments from complete solver costs. It passes $F \wedge \mathcal{A}$ directly to an exact OMT optimizer, without additional SMT calls to complete or validate the assignment. The same optimizer then continues under the original constraints.

Graph networks also support learning for satisfiability and integer optimization. NeuroSAT learns from satisfiability labels on a graph of literals and clauses [8]. Gasse et al. use a variable–constraint graph to learn branching decisions by imitating strong branching [9]. Neural diving predicts partial values for integer variables and solves the resulting smaller mixed-integer programs to complete them [10]. Our learning problem compares candidate partial assignments by the solver cost of obtaining and establishing an optimum. The representation must therefore describe both the predicates selected by a candidate and the truth values they receive, together with their logical and numerical relationships in the formula.

Cost-based algorithm selection learns which solver to use from measured performance [11]. Here, the alternatives are partial assignments generated for each formula, and their costs include both restricted optimization and subsequent exact search.

3 Method

NeurOMT generates candidate assignments, selects one with a GNN, and uses the selection in exact optimization, as shown in Figure 1. Generation limits the number of alternatives to score. The selector learns which restriction reduces the combined cost of the initial optimization and the remaining search. Exact continuation uses the original constraints to establish the final result.

3.1 Generating Candidate Assignments

Each candidate is built by choosing starting atoms and progressively assigning truth values to nearby atoms. We vary the starting point, the fraction of atoms to assign, and the proposed truth values. The candidate set $\mathcal{S}(I)$ also contains the empty assignment for direct solving.

Choosing a center. The formula graph connects the objective, variables, arithmetic atoms, and clauses. We use the three types of centers shown in Figure 2:

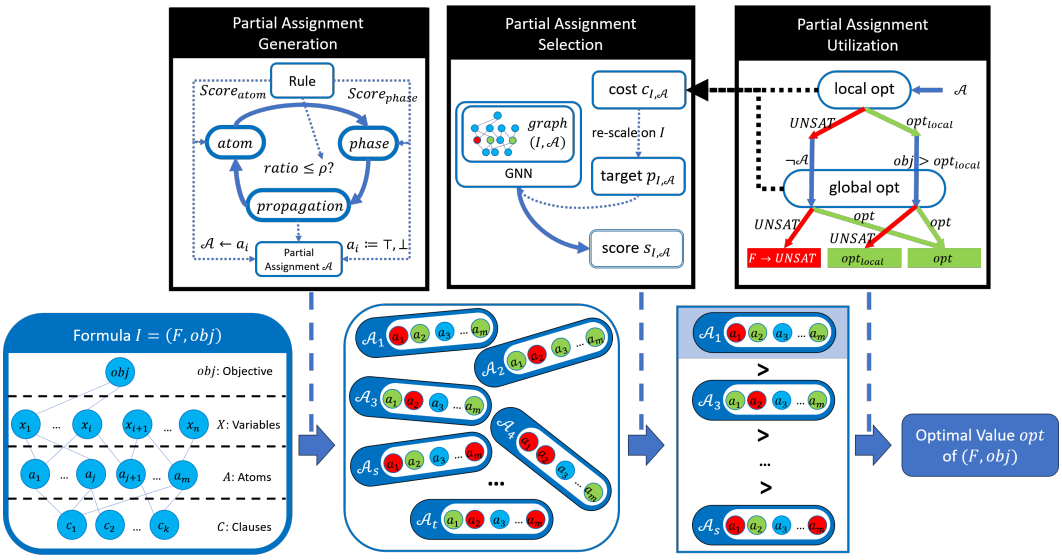

Figure 1: NEUROMT generates candidate partial assignments, ranks them with a GNN, and uses the selected assignment in exact optimization. Green, red, and blue atoms are true, false, and unassigned, respectively. Training uses the combined cost of restricted optimization and subsequent global search. The formula graph is shared across candidate scores.

- **Objective-centered.** We start from the atoms containing common variables with the objective.
- **Variable-centered.** We start from the atoms containing a selected variable. Variables with larger absolute objective coefficients are considered first, with their atom counts breaking ties.
- **Clause-centered.** We start from the atoms of a selected clause, considering shorter clauses first.

Capacity-aware and conflict-aware rules are two variants of variable-centered. The former rescales objective coefficients by resource consumption relative to capacity; the latter downweights variables whose atoms have many neighbors. These weights change which variables and atoms are considered first. The five rules share the same assignment procedure.

Scoring atoms and truth values. Let $p_{g,i}$ be the profit of variable x_i after rule-specific reweighting, and normalize it as $\bar{p}_{g,i} = p_{g,i} / \max_k |p_{g,k}|$. For $D_j = \{i : w_{ji} \neq 0\}$, the mean gain is $A_{g,j} = |D_j|^{-1} \sum_{i \in D_j} \bar{p}_{g,i}$. For a proposed truth value b, let $\bar{H}_{j,b}$ measure the clauses containing the corresponding literal, weighting each occurrence by the inverse clause length and normalizing across atoms. We score the two values by

$$S_g(a_j, \top) = -A_{g,j} + \bar{H}_{j,\top}, S_g(a_j, \bot) = A_{g,j} + \bar{H}_{j,\bot} \tag{4}$$

Positive mean gain favors making an upper-bound atom false; negative mean gain favors true. The clause term favors literals in shorter clauses, where fewer alternatives can satisfy the constraint. The normalized mean gain and clause coverage are added with equal weight. We prioritize atoms by $\max_b S_g(a_j, b)$ and try the higher-scoring truth value first.

Assigning and expanding. Construction begins with Boolean and integer-bound propagation. The starting atoms form a frontier: the atoms to consider next. At each step, we choose its highest-scoring unassigned atom. We try the preferred truth value on a copy of the current state and propagate its consequences. If it conflicts, we try the opposite value; if both conflict, we skip the atom. Successful assignments add neighboring atoms that share a variable or clause to the frontier. If the frontier is exhausted, we consider the remaining atoms. Growth stops at the requested fraction of

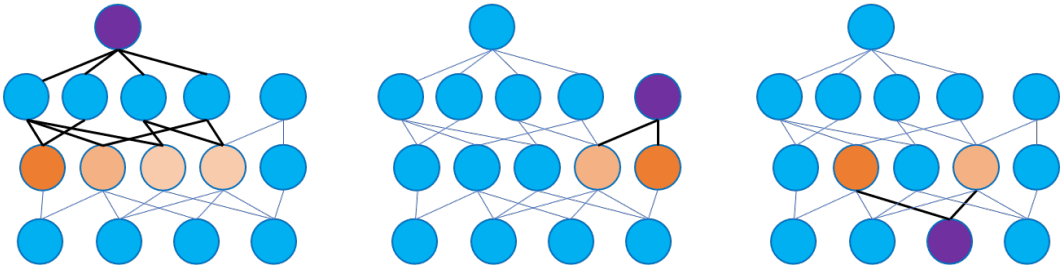

Figure 2: Objective, variable, and clause centers (left to right). Purple marks the center; orange marks relevant atoms, with darker shades indicating higher priority. The objective-centered rule starts from the atoms sharing variables with objective; variable-centered and clause-centered rules start from the atoms incident to their center.

initially unassigned atoms, counting propagated assignments, or when no atom remains. The propagation checks use Boolean units, integer bounds, and inconsistent proportional inequalities, without invoking an SMT solver.

Alternative truth values and candidate sizes. We also generate candidates using truth values suggested by a shared linear-programming relaxation. Each atom is evaluated at its fractional solution, and the suggested truth value goes through the same propagation checks. Different centers and assignment fractions, combined with the two ways to propose truth values, give 73 construction templates including direct solving. Identical assignments are merged before GNN scoring. Appendix A gives the priority formulas, normalization, and parameter settings; Appendix D gives the construction algorithm.

3.2 Learning to Select an Assignment

The selector assigns one score to each concrete candidate in $\mathcal{S}(I)$. Its input includes both the chosen atoms and their truth values, together with the formula in which the assignment will be used.

Encoding the formula. Following graph representations for neural OMT guidance [5], the graph in Figure 1 contains objective, variable, atom, and clause nodes. Objective–variable edges carry objective coefficients; variable–atom edges carry coefficients in the corresponding inequalities; atom–clause edges record positive or negative occurrences. Variable nodes connect predicates involving the same numerical unknown, while clause nodes connect predicates participating in the same disjunction.

Relation-specific message passing combines these numerical and logical relations into an embedding h_a for each atom. Thus an atom's representation reflects its own inequality, its connections to other predicates, and its relation to the objective. Pooling atom, clause, and objective embeddings, together with the instance context, gives a formula vector z_i. We encode each formula once and reuse these embeddings for all its candidates.

Representing a candidate. For candidate $\mathcal{A}_{ij}$, let B be its assigned atom set and B^+, B^- its true and false subsets. Write $\sigma_a = 1$ for true and $\sigma_a = -1$ for false. The candidate vector concatenates four means:

$$v_{ij} = [\mathrm{mean}_{a\in B}\, h_a \,;\, \mathrm{mean}_{a\in B}\, \sigma_a h_a \,;\, \mathrm{mean}_{a\in B^+}\, h_a \,;\, \mathrm{mean}_{a\in B^-}\, h_a]\,. \tag{5}$$

Empty pools are zero. The first component describes which atoms are selected; the other three preserve their truth values. Changing an atom from true to false changes its signed contribution and moves it between the two separate pools. The representation is independent of the order in which the generator assigned the atoms.

Scalar features u_{ij} record the assigned, true, and false fractions and summarize the candidate's construction sources, such as whether an LP proposal produced it. A candidate generated by several templates retains all these sources after deduplication. Rule identifiers are not model inputs. A

multilayer perceptron combines the formula and candidate representations:

$$s_{ij} = \mathrm{MLP}_\theta([z_i; v_{ij}; u_{ij}]), \qquad \pi_\theta(j \mid I_i) = \mathrm{softmax}_j(s_i). \tag{6}$$

The GNN uses mean aggregation over incoming messages. Section 4 also evaluates graph convolution and attention with the same candidate pooling and scoring head. Network dimensions, features, and aggregation details are given in Appendix B.

Supervision from solver cost. Offline, we execute every distinct training candidate through restricted optimization and exact continuation, as described in Section 3.3. The label uses the cumulative Z3 resource count R_{ij} for this complete execution. A candidate is therefore evaluated by the work needed to finish the entire optimization task, including the search after its restriction is removed. We normalize this count by $R_{i,\mathrm{bare}} > 0$ for direct solving of the same formula:

$$c_{ij} = \begin{cases} R_{ij}/R_{i,\mathrm{bare}}, & \text{if an optimum is established,} \\ \max\{2, R_{ij}/R_{i,\mathrm{bare}}\}, & \text{otherwise.} \end{cases} \tag{7}$$

Successful candidates with $c_{ij} < 1$ save solver work relative to direct optimization. Candidates that fail to establish an optimum receive a cost of at least two. Generation and inference time are measured separately in the runtime experiments.

Let $\Delta_{ij} = c_{ij} - \min_k c_{ik}$ and let q_i scale the cost gaps within an instance. We convert costs to a smoothed target distribution

$$p^*_{ij} = (1-\epsilon)\frac{\exp(-\Delta_{ij}/(\tau q_i))}{\sum_k \exp(-\Delta_{ik}/(\tau q_i))} + \frac{\epsilon}{m_i}, \qquad m_i = |\mathcal{S}(I_i)|. \tag{8}$$

Lower-cost candidates receive higher target probabilities; candidates with similar costs receive similar probabilities. The scale q_i normalizes the spread of costs within each instance, and ϵ assigns a small probability to every candidate. Training minimizes cross-entropy,

$$\mathcal{L}(\theta) = -\mathbb{E}_i \sum_j p^*_{ij} \log \pi_\theta(j \mid I_i). \tag{9}$$

Each instance contributes one loss term regardless of its number of unique candidates. Appendix C defines the gap scale.

Selection. Without calibration, the selector chooses $j_i^{\mathrm{top}} = \arg\max_j s_{ij}$. We also consider calibration against a reference assignment $\mathcal{A}_{i,r_i}$ using a margin $\eta \geq 0$:

$$\widehat{\mathcal{A}}_i = \begin{cases} \mathcal{A}_{i,j_i^{\mathrm{top}}}, & s_{i,j_i^{\mathrm{top}}} - s_{i,r_i} > \eta, \\ \mathcal{A}_{i,r_i}, & \text{otherwise.} \end{cases} \tag{10}$$

The reference is the empty assignment, which invokes direct solving. Validation determines the checkpoint and margin. Appendix C details calibration, and Appendix E gives the experimental settings. At inference, the generator constructs the candidates, the GNN encodes the formula once, and the scoring head ranks the resulting assignments. Only the selected assignment is sent to the optimizer; the costs of the other candidates are not measured.

3.3 Exact Optimization and Continuation

Given $\widehat{\mathcal{A}}$, the optimizer first solves $F \wedge \widehat{\mathcal{A}}$. If this problem has an attained optimum v, then v is a feasible objective value for the original problem. The assignment may have removed better solutions, so this first optimization does not by itself establish global optimality. We remove the assignment and search for a better solution by optimizing $F \wedge (obj > v)$. An infeasible continuation establishes v as globally optimal; otherwise the continuation supplies the global optimum. An infeasible restricted problem leads to direct optimization of F. The empty assignment also reduces to direct solving. All stages share a deadline, and an unfinished optimization returns an unresolved result.

Algorithm 1 summarizes the restricted solve, state restoration, and exact continuation.

Algorithm 1 Exact optimization with a selected partial assignment

Require: Formula F, objective obj, assignment $\widehat{\mathcal{A}}$, shared deadline T
Ensure: Global optimum, infeasible, or unresolved
$O \leftarrow$ INITIALIZEOPTIMIZER(F, obj)
if $\widehat{\mathcal{A}} = \emptyset$ **then**
    return OPTIMIZE(O, T)
PUSH(O); ADD$(O, \widehat{\mathcal{A}})$
$L \leftarrow$ OPTIMIZE(O, T)
POP(O)
if L is unresolved **then**
    return unresolved
if L has optimum v **then**
    ADD$(O, obj > v)$
$G \leftarrow$ OPTIMIZE(O, T)
if G is infeasible and L has optimum v **then**
    return v
return G

For example, let $0 \leq x, y \leq 2$, $x = 0 \vee y = 0$, and $obj = 2x + y$. The assignment $x \leq 0$ gives a restricted optimum of 2 at $(0, 2)$. Removing this assignment and adding $2x + y > 2$ allows the continuation to find $(2, 0)$, whose objective value is 4.

Proposition 1 (Exact continuation)**.** *If the invoked optimizations complete and each feasible optimum is finite and attained, this procedure returns the global optimum of F or proves F infeasible.*

Proof. When the restriction is empty or infeasible, the procedure optimizes F directly. Otherwise v is feasible for F, and $F \wedge (obj > v)$ retains every improving solution. If this continuation is infeasible, v is optimal. If feasible, its optimum exceeds every excluded value and is therefore globally optimal. □

Algorithm 1 uses one Z3 `Optimize` object. The selected literals are added in a temporary `push`/`pop` scope, so removing the candidate leaves the original constraints intact. The improving bound is then added to this restored state. The original formula's Boolean structure is handled within optimization, without a separate SMT call to complete or check the selected assignment. Complete execution time includes parsing, candidate generation, graph construction, and inference in addition to these optimization stages.

4 EXPERIMENTAL EVALUATION

We evaluate the mean-aggregation selector on knapsack and resource-constrained scheduling (RCSP). The experiments compare solver costs with direct Z3, examine variation across instances, and assess the contribution of graph encoding. All candidate-generation rules are enabled.

Data and protocol. Knapsack comprises 3,000 formulas with integer item multiplicities and logical side constraints, split into 1,600 training, 400 validation, and 1,000 test instances by parent instance. RCSP comprises 3,150 formulas from 3 templates of a cumulative scheduling model [12], with completion time, buffer storage and flow balancing as objectives. Each class has 40 training, 10 validation, and 1,000 test formulas. All variants at a shared processing capacity stay in the same split. Appendix E details the data and training settings.

Each instance is measured three times in randomized action order. All optimization stages share a 30-second budget. For NEUROMT, direct Z3 and OptiMathSAT, PAR-2 is elapsed time for a certified optimum within budget and 60 seconds otherwise. Solver-stage time includes backend preparation and parsing, optimization, and objective extraction.

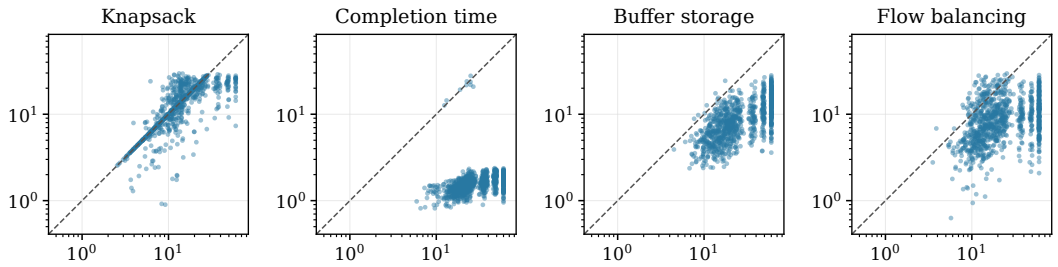

Figure 3: Solver-stage PAR-2 in seconds: direct Z3 (horizontal) versus NEUROMT (vertical), on logarithmic axes. Each point averages three repetitions; points below the diagonal favor NEUROMT.

4.1 MAIN RESULTS

Table 1 compares the same Z3 optimizer with and without learned selection. The selector reduces total solver-stage PAR-2 by 11.4% on knapsack, 94.4% on completion time, 70.2% on buffer storage and 65.2% on flow balancing. The guided solver certifies optima in all three repetitions on every knapsack instance and scheduling instance.

Table 1: NEUROMT with mean aggregation versus direct Z3. Z3 solved counts require certified optima within 30 seconds in all available runs; replacement instances have one run. The repaired cohort is conditioned on successful guided solves. Times are mean solver-stage PAR-2 in seconds; the ratio divides NEUROMT time by direct Z3 time.

Class	N	Z3 solved	Direct Z3	NEUROMT	Ratio
Knapsack	1000	920	15.929	14.112	0.886
Completion time	1000	579	31.254	1.745	0.056
Buffer storage	1000	564	32.863	9.808	0.298
Flow balancing	1000	670	27.992	9.749	0.348

Table 2 reports mean resource ratios on completed pairs of 1.002 for knapsack, 0.175 for completion time, 0.496 for buffer storage, and 0.418 for flow balancing. Both resource counts and elapsed time therefore decrease on the RCSP classes. On knapsack problems, only elapsed time decreases, indicating that NEUROMT favors harder instances when comparing with direct Z3. The resource statistic averages each instance's ratio, whereas the total PAR-2 ratio weights instances by their direct-solving cost and includes unsuccessful runs. They measure complementary aspects of solver performance.

Paired group-bootstrap 95% intervals for total solver-stage ratios are $[0.857, 0.916]$ on knapsack, $[0.052, 0.060]$ on completion time, $[0.290, 0.307]$ on buffer storage and $[0.332, 0.365]$ on flow balancing. All remain below one. Resampling preserves parent-instance and processing-capacity groups for each fitted model; Appendix F gives the procedure and complete intervals.

4.2 VARIATION ACROSS INSTANCES

Figure 3 shows lower solver-stage PAR-2 on 32.5% of knapsack, 99.5% of completion-time, 99.2% of buffer-storage, and 91.3% of flow-balancing instances. The corresponding median instance-wise ratios are 1.000, 0.059, 0.322, and 0.385. Solver savings extend across most scheduling instances. On knapsack, the median ratio remains 1.000 despite a lower aggregate cost, indicating that improvements are concentrated in the subset of harder instances.

Instance size. We partition each test set by variable count while keeping its trained model fixed. The cutoffs are 360 items for knapsack, 224 periods for completion time, and 60 periods for buffer storage; flow cutoffs are 120/76/64 periods for each subproblem. The knapsack subsets contain

628 smaller and 372 larger formulas, with solver-stage ratios of 0.918 and 0.845, confirming that NEUROMT earns more solver savings on harder knapsack instances. Each completion time subset contains 500 formulas, and ratios are 0.067 and 0.049. Buffer storage test set is divided into 503 smaller and 497 larger formulas, with ratios of 0.315 and 0.291. Flow balancing subsets contain 566 smaller and 434 larger formulas, with ratios of 0.414 and 0.301.

These results show solver savings on both smaller and larger instances within the test distribution. For RCSP, the two size groups correspond to different input templates, so their difference also reflects constraint structure. The comparison uses the same fitted selector in both groups; training and validation are unchanged. Table 4 reports the complete breakdown.

4.3 ABLATIONS

Selection validity ablation. We compare our GNN selection method with two variants: the best generation template fixed on each dataset, and a random-selection in all candidates. Table 5 compares their solver-stage costs. We fix 10 seeds for random-selection and report an average result on each dataset. On knapsack, only trained-GNN outperforms direct Z3, proving that the selection method learns to shift from generation rules considering different formulas. Dataset-fixed template shows potential improvements on both completion time and buffer storage.

Encoder ablation. The encoder variants share node features, candidate pooling, and the scoring head. Each is trained and calibrated using the same development partitions and protocol. The no-message control retains input projections and the scoring head while omitting graph propagation. Table 6 compares their solver-stage costs. Mean aggregation gives the lowest tested ratio on three datasets, while graph attention still performs better on completion time formulas.

4.4 END-TO-END COST

End-to-end (E2E) time adds input processing, candidate generation, graph construction, and inference with a preloaded model. For completion time, mean E2E PAR-2 is 13.917 seconds, corresponding to a ratio of 0.445 against direct Z3. The frontend adds approximately 12.172 seconds PAR-2 per instance against an average of 1.745 seconds for solver-only time, so most of the complete method's time is spent before exact optimization begins. For other datasets, the frontend ratio varies from 13.10% to 25.34%.

Table 7 also compares our E2E NEUROMT with OptiMathSAT on 100 instances per class sampled from those completed by direct Z3. OptiMathSAT achieves lower mean costs on knapsack and buffer storage, whereas NEUROMT achieves lower costs on completion time and flow balancing, with a maximal speedup of 3.747×.

Detailed time and resource measurements appear in Appendix F.

5 DISCUSSION AND CONCLUSION

NEUROMT learns to select candidate partial assignments using the cost of restricted optimization and subsequent exact search. Its optimizer retains the original formula and uses the selected restriction to obtain an objective bound. This separates the choice of a useful search region from the requirement to preserve a global optimum within that region.

The selector lowers total solver-stage PAR-2 on knapsack and flow balancing, comparing with the dataset-fixed best generation template. Mean aggregation gives the lowest tested knapsack cost, while solver savings comparing with direct Z3 persist without message passing on other datasets. Frontend cost outweighs solver savings in the knapsack experiments, and comparison with OptiMathSAT shows preference on formula features among solver backends. Reducing frontend costs and employing different solver backends for corresponding problems remain directions for further work.

AI USE STATEMENT

AI tools assisted with manuscript organization, language editing, code inspection, and consistency checks between the text and reported results.

REFERENCES

[1] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 607–618. ACM, 2014.

[2] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ—an optimizing SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

[3] Roberto Sebastiani and Patrick Trentin. Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 335–349. Springer, 2015.

[4] Gabriele Masina and Roberto Sebastiani. Exploiting partial-assignment enumeration in optimization modulo theories. In *Frontiers of Combining Systems*, pages 117–134. Springer, 2025.

[5] Justin Wong, Pei-Wei Chen, Tianjun Zhang, Joseph Gonzalez, Yuandong Tian, and Sanjit A. Seshia. Ashera: Neural guided optimization modulo theory. Technical Report UCB/EECS-2023-103, EECS Department, University of California, Berkeley, 2023.

[6] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract DPLL procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[7] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 1267–1329. IOS Press, 2nd edition, 2021.

[8] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[9] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2019.

[10] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[11] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[12] Antton Kasslin and Jeremias Berg. An optimization modulo theories-based approach to cumulative scheduling with delays. In *Proceedings of the 23rd International Workshop on Satisfiability Modulo Theories*, pages 16–28, 2025.

A CANDIDATE GENERATION DETAILS

This section specifies the weighted structural generator. A source template $r = (g, s, \rho, lp)$ selects a priority group, a seed, a requested coverage, and a phase provider. Let $U \subseteq A$ be the selectable atoms not fixed by root propagation. Two atoms in U are neighbors if they share a variable or a clause; $N(a)$ excludes a itself. All clause statistics below use the original clause representation, before candidate-specific propagation.

Priority vectors. Write $obj = \boldsymbol{v}^\top \boldsymbol{x}$ and $a_j : \boldsymbol{w}_j^\top \boldsymbol{x} \le b_j$. Objective-, variable-, and clause-guided priorities use $p_{g,i} = v_i$. For capacity-guided priorities, define

$$d_i = \sum_{k \in K} \frac{w_{ki}}{b_k}, \qquad \bar{d} = \frac{1}{n} \sum_i d_i, \qquad p_{g,i} = \frac{v_i}{d_i}.$$

Here K indexes arithmetic atoms occurring as positive unit clauses with $b_k > 0$ and nonnegative coefficients. For $d_i = 0$, use $p_{g,i} = v_i$ to avoid division by zero. Conflict-guided priorities use

$$p_{g,i} = \frac{v_i}{\max\{1, \sum_{a \in U} \mathbf{1}(w_{ai} \neq 0)\}}.$$

These vectors determine generation priorities only; the exact objective remains obj.

Normalize profits by $\bar{p}_{g,i} = p_{g,i} / \max_k |p_{g,k}|$. Let $D_j = \{i : w_{ji} \neq 0\}$. Define mean gain and signed clause coverage by

$$A_{g,j} = \frac{1}{|D_j|} \sum_{i \in D_j} \bar{p}_{g,i},$$

$$H_{j,p} = \sum_{c \in C:\, \ell(a_j,p) \in c} \frac{1}{|c|}, \qquad \bar{H}_{j,p} = \frac{H_{j,p}}{\max\{1, \max_{a_k \in U,q} H_{k,q}\}}.$$

Set all normalized profits to zero if their denominator is zero, and set $A_{g,j} = 0$ if D_j is empty. The signed literal is $\ell(a, \top) = a$ and $\ell(a, \bot) = \neg a$. The truth-value priorities follow Equation 4; the atom priority is $S_g(a_j) = \max_{p \in \{\top, \bot\}} S_g(a_j, p)$. Mean gain and normalized clause coverage enter each phase score with equal weight.

Seeds and deterministic ordering. The objective group starts from atoms containing common variables with the objective. The variable, capacity, and conflict groups start from active atoms incident to a seed variable; variables are ordered by decreasing $|p_{g,i}|$, decreasing active-atom incidence count, and increasing variable index. Clause seeds contain the active atoms of a clause, with clauses ordered by length and then their atom-index sequence. Empty and duplicate seed sets are skipped. For atom selection, ties in $S_g(a)$ prefer more neighbors and then a smaller atom index. Objective-seed alignment ties first prefer larger $S_g(a)$, then use the same atom tie breakers. Greedy phase ties prefer true.

One objective seed and two seeds for each other group are combined with $\rho \in \{0.25, 0.5, 0.75, 0.9\}$ and both phase providers. Including bare solving gives $1 + (1 + 2 + 2 + 2 + 2) \times 4 \times 2 = 73$ source templates. An unavailable seed or empty construction maps to the empty candidate. Deduplication uses the complete sorted $(\text{atom}, \text{phase})$ signature and preserves all source aliases of each unique candidate.

Propagation state. A state contains an atom-to-phase map and integer-variable bounds. The root state incorporates unit clauses and phases implied by variable intervals. Each extension copies this state, commits a proposed literal, and propagates Boolean units, integer interval implications, and single-variable bound tightening. Contradictory phases, falsified clauses, empty domains, and inconsistent proportional linear atoms reject the trial. Returning a failed trial does not change the parent state. The resulting filter is incomplete for SMT feasibility; its purpose is to reject detectable conflicts cheaply.

Coverage counts all assigned atoms in U, including implications of explicit decisions. Root-implied literals remain in the propagation state but are excluded from the returned cube. Growth stops after assigning at least $\lceil \rho |U| \rceil$ active atoms, or after exhausting all unassigned, unrejected active atoms. Propagation may exceed the requested coverage.

LP phase provider. The LP relaxation retains variable domains, unit arithmetic constraints, and clauses expressible as binary-variable literals. Other disjunctions are omitted. For integer coefficients, a negated atom $\neg(\boldsymbol{w}^\top \boldsymbol{x} \le b)$ is converted to $-\boldsymbol{w}^\top \boldsymbol{x} \le -b-1$ before integrality is relaxed. The provider first optimizes the original objective over this relaxation. If unbounded, it solves a zero-objective feasibility LP instead. A single proposal-generation process supplies all LP-guided templates of an instance. LP proposals are computed with HiGHS using one thread and a two-second time limit per LP call.

Arithmetic atoms are evaluated at the fractional point without rounding its variables, with comparison tolerance $10^{-9}\max\{1, |\boldsymbol{w}_j^\top \boldsymbol{x}|, |b_j|\}$. The resulting phases pass through the same propagation trials as greedy phases. Different implications may therefore produce different final supports. If no point is available, the LP-guided template maps to the empty candidate.

B Graph Representation and Scoring Details

Graph inputs. The graph has objective, variable, atom, and clause nodes and bidirectional objective–variable, variable–atom, and atom–clause relations. Their input feature dimensions are 3, 4, 8, and 5, respectively. Objective–variable and variable–atom edges carry two numerical features; atom–clause edges carry one polarity feature. Root Boolean propagation determines the residual atom–clause edges. A separate four-dimensional context is

$$(\log(1+n_{\mathrm{var}}), \log(1+n_{\mathrm{atom}}), \log(1+n_{\mathrm{clause}}), 0),$$

where the last entry is the normalized assignment depth, zero at initial selection. Its projection is passed to the scoring head rather than through graph message passing.

Node and edge features. Let $L(t) = \log(1+t)$, $s = \max\{1, \max_i |v_i|\}$, and $r_j = \max\{1, \sum_i |w_{ji}|\}$. Write $n = |X|$, $m = |A|$, and $q = |C|$. Let d_i count arithmetic atoms containing x_i, and let o_j^+, o_j^- count positive and negative occurrences of a_j, with $o_j = o_j^+ + o_j^-$. Root Boolean unit propagation supplies $t_j \in \{-1, 0, 1\}$ for false, unassigned, and true. The node features are

$$\begin{aligned} f_{\mathrm{obj}} &= [L(s), L(n), L(q)], \\ f_{x_i} &= [v_i/s, \mathbf{1}[v_i > 0], L(d_i), L(D_i)], \\ f_{a_j} &= [b_j/r_j, L(r_j), L(o_j), (o_j^+ - o_j^-)/\max(1, o_j), t_j, 0, 0, 0], \\ f_c &= [L(|c|), |c|/\max(1, m), p_c, s_c, u_c]. \end{aligned}$$

Here D_i is the finite domain width, the absolute finite endpoint for a one-sided domain, or zero for an unbounded domain. The clause quantities p_c, s_c, u_c are the positive-literal fraction, root-satisfaction indicator, and root-unassigned fraction. Objective–variable edges use $[v_i/s, L(|v_i|)]$; variable–atom edges use $[w_{ji}/r_j, L(|w_{ji}|)]$. Reverse edges reuse these features. Atom–clause edges encode positive and negative occurrences as 1 and 0. Root-satisfied clauses have no atom edges; other clauses connect only unassigned literals. Occurrence counts and node features use the original clauses.

At embedding width $h = 48$, the formula vector z_i has dimension $6h$: atom mean and maximum, clause mean and maximum, objective mean, and the projected context. The candidate vector v_{ij} has dimension $4h$, containing the selected-atom mean, phase-signed mean, true-atom mean, and false-atom mean. Empty pools are zero. The first two pools divide by the number of selected atoms; the latter two divide by their respective phase counts.

The nine scalar candidate features comprise assigned/true/false fractions of all formula atoms and six alias summaries: presence of LP and non-LP sources, absence of non-bare sources, the fraction of catalog templates merged into the candidate, and minimum/maximum requested coverage among non-bare sources. Coverage summaries are zero if there is no non-bare source. Rule and seed identities are not direct inputs. Thus the scoring head takes $6h + 4h + 9 = 489$ features and applies linear layers of widths $489 \to 48 \to 48 \to 1$, with GELU after the first two layers.

Mean aggregation. The encoder uses three message-passing layers. Each node type has an input projection consisting of a linear map, GELU, and layer normalization. For an edge $u \xrightarrow{r} v$ with

feature vector e_{uv}, layer t forms

$$m_{uv}^{(t)} = W_r^{(t)}[h_u^{(t)}; e_{uv}] + b_r^{(t)}, \qquad \bar{m}_v^{(t)} = \frac{\sum_{u \xrightarrow{r} v} m_{uv}^{(t)}}{\max(1, d_v)},$$

where d_v counts all incoming edges across relations. A node-type-specific two-layer MLP, with an intervening GELU, updates the embedding through

$$h_v^{(t+1)} = \mathrm{LayerNorm}\big(h_v^{(t)} + \mathrm{MLP}_{\mathrm{type}(v)}^{(t)}(\bar{m}_v^{(t)})\big).$$

Relation-specific message maps are shared across edges of the same type.

C Training and Calibration Details

The resource label uses the optimizer's final cumulative resource counter, including operations between the restricted and continuation calls. For the complete-execution cost labels c_{ij} defined in the main text, the target scale is

$$\Delta_{ij} = c_{ij} - \min_k c_{ik}, \qquad q_i = \max\{0.05, Q_{0.75}(\{\Delta_{ik}\}_{k=1}^{m_i})\}, \tag{11}$$

using a linearly interpolated sample quantile. The soft-target loss uses temperature $\tau > 0$ and smoothing $0 \le \epsilon < 1$. One cross-entropy term is averaged per instance, so instances with more unique candidates do not acquire a larger weight solely from their candidate count. Training settings are given in Appendix E.

Margin calibration. Let r_i index the empty candidate, which invokes direct optimization, and let $\mu_i = s_{i,j_i^{\mathrm{top}}} - s_{i,r_i}$. The calibration routine enumerates

$$\mathcal{H} = \{0\} \cup \{\mu_i : i \in \mathcal{V}\} \cup \{1 + \max_{i \in \mathcal{V}} \mu_i\},$$

where $\mathcal{V}$ is the validation set, and applies the strict switch test $\mu_i > \eta$. It minimizes mean validation cost over $\eta \in \mathcal{H}$; ties prefer more switches, then the smaller threshold. The numerical threshold is frozen for inference.

D Detailed Candidate Generation

The following procedures make state changes and termination conditions explicit. They apply to the finite, attained, single-objective LIA setting. Write $Q = (D, B)$ for a propagation state containing assignments D and bounds B. The procedure $\mathrm{TryExtend}(Q, a, p)$ starts from an independent copy of Q and either returns a propagated state or reports conflict; failure never mutates Q. A missing seed, unavailable required LP proposal, or failed root propagation returns the empty candidate.

Construct one source candidate. Given root state Q_0, active set U, seed frontier f, and priorities S_g:

1. Set $Q \leftarrow Q_0$, rejected set $J \leftarrow \emptyset$, and $k \leftarrow \lceil \rho|U| \rceil$. If $U = \emptyset$, the seed is unavailable, or a required LP proposal is unavailable, return $\emptyset$.
2. If $|U \cap \mathrm{dom}(D)| \ge k$, go to Step 6. Form $V \leftarrow U \setminus (\mathrm{dom}(D) \cup J)$; if $V = \emptyset$, go to Step 6.
3. Choose the highest-priority atom from $V \cap f$ if this set is nonempty, and otherwise from V. Let p be its LP hint or greedy preferred phase, according to the source template.
4. Set $Q' \leftarrow \mathrm{TryExtend}(Q, a, p)$. If this fails, set $Q' \leftarrow \mathrm{TryExtend}(Q, a, \neg p)$. If both fail, add a to J and continue with Step 2.
5. Otherwise set $Q \leftarrow Q'$ and $f \leftarrow f \cup \bigcup_{b \in U \cap \mathrm{dom}(D)} N(b)$. Continue with Step 2.
6. Return the sorted signature $\mathcal{A} = \{(a, D(a)) : a \in U \cap \mathrm{dom}(D)\}$.

Every iteration either assigns an available atom or adds it to J; hence there are at most $|U|$ iterations. Propagation retains previous assignments, so its implications only accelerate exhaustion or attainment of the target.

Generate and select. Compute one shared LP proposal for the instance when required, and initialize the root propagation state for each source construction. Canonicalize the resulting signatures together with $\emptyset$, retaining an alias list for each element of $\mathcal{S}(I)$. Encode the formula once and score all members of $\mathcal{S}(I)$. Compare the highest-scoring assignment with the empty candidate using the calibrated margin to obtain $\widehat{\mathcal{A}}$, then execute Algorithm 1.

E Experimental Protocol

Benchmarks and partitions. Knapsack formulas have 193–391 integer variables representing item multiplicities, with capacity constraints and logical side conditions. The 3,000 formulas are partitioned into 1,600 training, 400 validation, and 1,000 test instances. A parent instance and all its variants remain in one partition. Every evaluated method uses the same test set.

The scheduling benchmarks use three templates of a cumulative scheduling model [12]. Completion time minimizes makespan; buffer storage minimizes cumulative storage requirements. Flow balancing maximizes the minimum flow (MAXMIN), minimizes the maximum flow (MINMAX), or minimizes the difference between them (MINDIFF). The number of integer variables varies from 1,118 to 1,490 in completion time templates, 290 to 458 in buffer storage and 244 to 866 in flow balancing. Each objective class has 40 training, 10 validation, and 1,000 test formulas.

Training and selection. The mean-aggregation selector uses all candidate-generation rules and is initialized by knapsack pretraining. Scheduling models adapt this initialization using their 40 target training formulas. Encoder variants use the same training and validation partitions, initialization protocol, loss, and random seed. Training runs for 30 epochs at learning rate 10^{-4}. Validation selects the checkpoint, including the initial checkpoint, and the switching margin against direct solving. The selected weights and margin are then fixed for test evaluation.

Timing and metrics. Each instance has three repetitions with randomized action order. Restricted optimization and continuation share a 30-second deadline. A certified optimum receives its measured time; an unsuccessful repetition receives a PAR-2 penalty of 60 seconds. We average these values within each instance. For method costs t_i and direct-Z3 costs b_i, the total ratio is $\sum_i t_i / \sum_i b_i$, while the mean instance ratio is $\frac{1}{N} \sum_i t_i / b_i$. Solved counts require success in every repetition.

Solver-stage time includes backend preparation, parsing, optimization, and objective extraction. Frontend time includes input processing, candidate generation, graph construction, and selection. It is measured once per instance and added to each solver repetition. E2E PAR-2 applies the same 30-second deadline to this sum. Model loading and offline training are excluded. Measurements use single-threaded workers. Resource ratios average the cumulative Z3 counter ratio over pairs for which both methods certify optima in all three repetitions.

F Additional Results

Table 2 reports both time aggregations and resource ratios. The two scheduling classes also have E2E measurements.

Table 2: NeurOMT versus direct Z3. Solved counts require solver completion in all available runs; replacement instances have one run. R is the mean resource ratio over the P completed pairs with compatible recorded counters. Time columns give the mean of instance-wise solver-stage PAR-2 ratios, the ratio of summed solver-stage PAR-2, and the corresponding E2E ratio.

Class	N	Solved	R	P	Mean time	Total time	E2E
Knapsack	1000	1000	1.002	920	0.982	0.886	1.019
Completion time	1000	1000	0.175	579	0.069	0.056	0.445
Buffer storage	1000	1000	0.495	564	0.357	0.298	0.360
Flow balancing	1000	1000	0.418	670	0.475	0.348	0.466

Variation across test groups. Table 3 gives per-instance wins, median ratios, and paired 95% bootstrap intervals. We resample the 500 parent-instance groups for knapsack and the 500 capacity groups for each scheduling class with replacement. Each draw retains all formulas in a sampled group and the paired costs of both methods. We recompute the ratio of summed PAR-2 for 10,000 draws and use its 2.5th and 97.5th percentiles. The intervals measure variation across test groups for the fitted models.

Table 3: Paired comparison of the mean-aggregation selector with direct Z3. Wins count lower solver-stage PAR-2; the median is the instance-wise ratio. Intervals are 95% paired group-bootstrap intervals for total ratios (10,000 draws), preserving original parent groups for knapsack and source-window template groups for scheduling; intervals condition on the repaired cohort. A dash indicates an unavailable measurement.

Class	Wins/N	Median	Solve interval	E2E interval
Knapsack	325/1000	1.000	$[0.857, 0.916]$	$[0.984, 1.057]$
Completion time	995/1000	0.059	$[0.052, 0.060]$	$[0.431, 0.461]$
Buffer storage	992/1000	0.322	$[0.290, 0.307]$	$[0.350, 0.371]$
Flow balancing	913/1000	0.385	$[0.332, 0.365]$	$[0.446, 0.488]$

Performance by instance size. Table 4 partitions the same test instances at 360 items for knapsack, 224 periods for completion time, and 60 periods for buffer storage; flow cutoffs are 120/76/64 periods for MAXMIN/MINMAX/MINDIFF. Each model is evaluated in both size groups without retraining. The scheduling groups correspond to the three input templates. These are descriptive test-set comparisons; they use the training and validation partitions specified above.

Table 4: Solver-stage cost by test-instance size. Ratios compare summed PAR-2 with direct Z3.

	Smaller		Larger	
Class	N	Ratio	N	Ratio
Knapsack	628	0.918	372	0.845
Completion time	500	0.067	500	0.049
Buffer storage	503	0.315	497	0.291
Flow balancing	566	0.414	434	0.301

Selection validity ablation. Table 5 compares the solver-stage costs for GNN, dataset-fixed best template, and random selection in all candidates. 10 seeds are fixed for random-selection, and we use an average as its result.

Table 5: Selection ablation comparing GNN selection results with the best generation template and a 10-seed random-selection average. Entries are total solver-stage PAR-2 ratios to direct Z3.

Encoder	Knapsack	Completion time	Buffer storage	Flow balancing
GNN	0.886	0.056	0.298	0.348
Best Template	1.062	0.048	0.290	0.390
Random	1.241	0.883	0.788	0.750

Encoder ablation. Table 6 compares the solver-stage costs for the encoder variants. They share node features, candidate pooling, and the scoring head. Each is trained and calibrated using the same development partitions and protocol. The no-message control retains input projections and the scoring head while omitting graph propagation.

E2E NEUROMT, Z3 and OptiMathSAT. Table 7 compares our E2E NEUROMT with OptiMathSAT on 100 instances per class sampled from those completed by direct Z3.NEUROMT achieves lower costs on completion time and flow balancing, with a maximal speedup of 3.747×.

Table 6: Encoder ablation with all generation rules. Entries are total solver-stage PAR-2 ratios to direct Z3.

Encoder	Knapsack	Completion time	Buffer storage	Flow balancing
Mean aggregation (ours)	0.886	0.056	0.298	0.348
Graph convolution	1.006	0.397	0.473	0.413
Graph attention	1.007	0.051	0.308	0.431
No message passing	0.989	0.288	0.334	0.451

Table 7: End-to-end comparison on 100 instances per class sampled uniformly from the current test instances completed by direct Z3. Times are mean E2E PAR-2 in seconds, with a 30-second deadline and a 60-second failure penalty. For direct solvers, costs include backend preparation and solving; NEUROMT additionally includes frontend processing and selection.

	OptiMathSAT		Direct Z3	NEUROMT
Class	Solved	E2E PAR-2	E2E PAR-2	E2E PAR-2
Knapsack	87/100	13.575	13.794	16.149
Completion time	29/100	48.072	19.559	12.829
Buffer storage	100/100	1.282	17.227	9.021
Flow balancing	75/100	19.872	14.940	11.920