Ashera; Neural Guided Optimization Modulo Theory

Justin Wong
Pei-Wei Chen
Tianjun Zhang
Joseph Gonzalez
Yuandong Tian
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-103
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-103.html

May 11, 2023

Acknowledgement

This research is supported in part by NSF CISE Expeditions Award CCF-1730628, NSF NRI \#2024675 and under the NSF AI4OPT Center. UC Berkeley research is also supported by gifts from Alibaba, Amazon Web Services,
Ant Financial, CapitalOne, Ericsson, Meta, Futurewei, Google, Intel, Microsoft, Nvidia, Sco
tiabank, Splunk and VMware.

Ashera: Neural Optimization Modulo Theory

Justin Wong*[1], Pei-Wei Chen*[1], Tianjun Zhang[1],
Joseph E. Gonzalez[1], Yuandong Tian[2], Sanjit A. Seshia[1]
[1]University of California, Berkeley [2]Meta AI Research

Abstract

Applications of Satisfiability Modulo Theories (SMT) within design automation and software/hardware verification often require finding models whose quantitative cost objective is guaranteed to be optimal. As an example, in worst-case execution time analysis, it does not suffice to simply discover a feasible execution trace; we are instead interested in proving properties on the longest execution trace. Such problems can be formulated as Optimization Modulo Theory (OMT), and solving them is much more challenging than both SMT problems and unconstrained optimization. Current solutions struggle to scale to problems of large size, because they require experts to tune solvers and carefully craft problem encodings. This approach is not only problem-specific but also requires manual effort. Recent progress in neural techniques have been successfully applied to Mixed Integer Linear Programming (MILP) and certain instances of the Traveling Salesman Problem (TSP). We make the case for learning-based solvers in OMT and present Ashera, a neural-guided OMT solver. Ashera innovates on prior art by introducing Logical Neighborhood Search and neural-based warm starting. Additionally, we introduce new benchmarks for learning-based OMT techniques, targeted at real-world applications including scheduling and multi-agent TSP. Ashera exhibits as much as a 3x speedup and shows improved scaling compared to MILP approximation as used in industry and state-of-the-art OMT solvers.

1 Introduction

Analysis of worst-case execution time (WCET) of software and hardware requires not only identifying valid executions of a program but also finding a provably slowest execution. Existing solvers for Satisfiability Modulo Theories (SMT) have found remarkable practical success in providing guarantees on previously feasibility problems in domains such as software and hardware verification (Dutertre and De Moura 2006; Moura and Bjørner 2008; Leino 2010; Katz et al. 2017). However, there has been limited progress on Optimization Modulo Theories (OMT) problems like WCET (Henry et al. 2014) that require provably cost-optimal solutions not just feasible solutions. In these scenarios, ranging from software security(Bertolissi, Dos Santos, and Ranise 2018; Henry et al. 2014) to scheduling and planning(Leofante et al. 2017; Kovásznai, Biró, and Erdélyi 2017), require OMT to support an objective function.

While OMT is a more general optimization framework, solving it becomes more difficult. There are two major challenges. First, existing approaches to OMT either use the Big M approximation to reduce the problem to Integer Linear Programming ILP (Gurobi Optimization, LLC 2021) or do iterative calls to SMT (Sebastiani and Trentin 2015; Bjørner, Phan, and Fleckenstein 2015), which scales poorly. Second, the solvers are designed to be general purpose and problem agnostic. To address poor scalability in a per application basis, experts manually tune hyperparameters and problem encodings which often improves the solving time from days to a matter of minutes. Such solutions, while feasible, are often expensive and time-consuming, and ignore the past experience of solved problems.

In this paper, we present a codesigned neural OMT solver, called *Ashera* to address these two challenges by introducing two components: *an OMT engine* and *a neural diver*.

First, our OMT engine in Ashera iterates between calling an SMT solver and an optimality-aware theory solver, TS*. Because we use SMT to decompose disjunctions into guided search over conjunctive cases, we can avoid using Big M. In SMT, theory solvers (TS) are domain specific subsolvers for a conjunction of constraints. For instance, for LIA theory specific properties to prune and speed up the enumeration such as $x > 0 \land x < 1$ indicates no integer x is feasible. We introduce *optimality-aware theory solver*, TS*, which additionally has awareness of the optimization objective. More concretely for Linear Integer Arithmetic (LIA), we use Z3 for quantifier-free LIA SMT and Gurobi for ILP.

Second, our neural diver learns to directly predict promising partial assignments that lead to a better cost when extended to full assignments. We do this by training a Graph Neural Network (GNN) on existing solutions of related problems, extracting the knowledge from past experience. Then, the same SMT solver used in our OMT engine is used to check the validity of these candidate assignments to guarantee soundness. Our approach is inspired by the recent success of (Nair et al. 2020) that applies similar idea to Mixed Integer Linear Programming (MILP). The efficiency of existing solvers can be substantially improved by transferring knowledge of solving one problem instance to another similar one. We observe that in practice, optimiza-

*These authors contributed equally.

tions are done repeatedly on similar instances drawn from an underlying distribution, for example on a monthly basis for network planning (Zhu et al. 2021). As such, the problems to be solved regularly differ only slightly, but the optimization procedure needs to be redone, as modified constraints change the solution space. As a result, learning from previous solved problems can speed up the next optimization without any manual human input. Our approach differs from existing work that learn end-to-end a solver to replace existing systems (e.g., learned query optimization (Yang et al. 2022) and chip placement (Mirhoseini et al. 2020)). In contrast, we co-design our OMT approach to contribute a new generalized solver while exploiting opportunities for learning-based components.

As existing benchmarks provide dozens of diverse OMT problems, we build a benchmark for evaluating learning-based OMT solvers. Our benchmark provides over $58,000$ OMT problems from two families of problems: task scheduling and multi-agent Traveling Salesman Problem (TSP). Using this benchmark we demonstrate that Ashera can learn on problems with less variables and constraints and generalize to larger problems within the same problem class.

Our work makes the following contributions:

- A benchmark for evaluating learning-based OMT solving with problem families in scheduling and multi-agent traveling salesman problems.
- To our knowledge, we present the first learning-based OMT solver in Ashera. The solver focuses on support for Linear Integer Arithmetic (LIA), but the framework generalizes to other theories.
- For scheduling, Ashera in contrast is **3x faster** and solves **three more problems** than the widely-used commercial solver, Gurobi, on problems with 10 and 11 tasks, respectively. OptiMathSAT and Z3 are unable to solve any problems of 11 tasks within 1 hour timeout.
- Ashera is **18% faster** than OptiMathSAT on multiagent TSP with 15 waypoints, where Gurobi and Z3 timeout.

2 Related works

2.1 Solvers for OMT

The last decade has produced two prominent Optimization Modulo Theory (OMT) solvers: 1) OptiMathSAT (Sebastiani and Trentin 2015), which applies a binary search approach to discovering the optimal solution, and 2) *νZ* (Bjørner, Phan, and Fleckenstein 2015), which iteratively uses SMT to find a strictly better feasible solution and locally improve it by coordinate-wise search for strictly improving boundary solutions. OptiMathSAT further uses sorting networks to decouple the Boolean reasoning from the arithmetic solving. *νZ* exploits carefully engineered MaxSMT and pseudo Boolean solvers to provide competitive performance when the OMT problem lies within these domains. Neither of these works perform well at scale and applications are largely dominated by ILP approximations.

Similar to OMT, Inez (Manolios, Pais, and Papavasileiou 2015), a solver for Mathematical Programming Modulo Theory, integrates ILP solvers with solvers for first order theories. However, Inez relies on the Big M encoding or built in constraint handlers in ILP solvers to reason about disjunctions and instead focus on extending ILP solvers to support uninterpreted functions and support for user-provided axioms. Ashera explicitly reasons about disjunctions arising from the logic structure of OMT problems.

2.2 Neural Guidance for Combinatorial Optimization

The Integer Linear Program (ILP) is a well-studied class of optimization problems in large part due to its prominence in operations research, computer vision, scheduling, and other domains. Branch and bound, one particular tree search algorithm, is the best known approach for solving ILPs. Branch and bound is able to incrementally establish a lower and upper bound via a feasible point and a solution to a linear relaxation of the ILP respectively until tightness is achieved.

In (Balcan et al. 2018), the authors propose to replace empirical heuristics for selecting variables to branch with a data-driven methodology that achieves provable complexity bounds. The authors show that they can learn a convex combination of scoring rules (which each determine the ordering of node branching) that is nearly optimal in expectation over a distribution of original ILPs. Furthermore, the optimization process over scoring rules can be seen as performing empirical risk minimization (ERM) of the original optimization objective subject to the input variable constraints.

Another approach taken by Wu et al. (Wu et al. 2021a) is to train a model to reconstruct locally optimal solutions. This model is trained by taking feasible solutions and resolving the optimization with a random subset of variables masked out. The model learns how to improve solutions locally and recognize strategies that generalize to adjacent regions in the feasible set. However, this approach fails to recognize that due to combinatorial explosion subproblem optimization is often negligible and the real challenge is identifying and proving optimal the global optima not local optimas.

The approach taken by Nair et al. (Nair et al. 2020) extends Balcan et al. (Balcan et al. 2018)'s approach with a neural diver, which takes the bipartite graph representation of variables and constraints to predict plausible partial assignments. These partial assignments can then be explored in parallel by instantiating Mixed Integer Linear Programming (MILP) instances over a smaller variable space. Our neural diver extends Nair et al. to support OMT problems.

2.3 Neural Guidance for Combinatorial Applications

Using neural networks to solve TSP (Bello et al. 2016) has been extensively studied dating back to the development of Hopfield neural networks in 1985 (Hopfield and Tank 2004). More recent efforts such as Selsam et al. (Selsam et al. 2019) have attempted to learn end-to-end models for SAT solvers.

Alternatively, a presolve phase is employed before solving to explore promising regions. This approach is exemplified by NeuroPlan (Zhu et al. 2021), which applies neural guidance to large-scale network planning problem. These problems often takes days or weeks for integer linear programming (ILP) to find even a feasible solution. For this, NeuroPlan learns an RL agent to predict a good initial solution to large-scale network planning problem modeled as ILP. The RL agent constructs the solution progressively by picking which network connection to be used to increase the capacity between two nodes to satisfy the communication requirements, and verify the feasibility via efficient checking techniques, reusing previous computational results. Once the initial solution is found, a follow-up ILP solving becomes much faster.

3 Preliminaries

In this section, we provide some background on OMT and graph neural network architectures used in Ashera.

3.1 Optimization Modulo Theories (OMT)

Optimization Modulo Theories (OMT) extends SMT, guaranteeing that an optimal feasible solution is returned. In addition to the satisfaction formula, an OMT problem includes an objective function C which maps assignments to a total ordered set. When the domain of C is real or floating point, a tolerance δ must be specified.

A Satisfiability Modulo Theory (SMT) problem decides the satisfiability of a first-order formula within a theory (Barrett et al. 2009). We focus on the theory of Linear Integer Arithmetic (LIA), but note that SMT and OMT extend to other theories, for instance, arrays and strings. For LIA, consider *atomic formulas* as linear inequalities of the form: $atom_i \triangleq \vec{a_i} \cdot \vec{x} \bowtie b_i$ where $\bowtie \ \triangleq \{<, \leq, >, \geq, =\}$. These atoms may differ when considering different theories, and we denote a theory *literal* as an atomic formula or the negation of one.

We build up *clauses* and subsequently *formulas* in Conjunctive Normal Form (CNF) from these atoms as

$$clause_j \triangleq \bigvee_i literal_i \qquad formula \triangleq \bigwedge_j clause_j.$$

With the standard interpretation of atoms, a model or assignment that satisfies a formula is one where the evaluation of the formula is True. We denote partial assignments as α and full assignments as $\mathcal{A}$.

We define a *Boolean backbone* $\mathcal{B}$ as the set of literals where the polarity of each literal depends on the truth value of the corresponding atom when applying a full assignment $\mathcal{A}$, i.e. a literal is in positive polarity if the corresponding atom valuates to true. A Boolean (backbone) assignment is the truth value assignment to all literals in the formula. Note that an assignment uniquely specifies a Boolean backbone but multiple assignments can share a common backbone.

Lazy SMT solves with a two stage iterative process. First, a SAT solver identifies a candidate Boolean assignment, viewing clauses as simple Boolean functions. Then, once a Boolean assignment is chosen, the formula can be expressed as a conjunct of atomic theory constraints. As such, the constraints can then be passed along to a specialized theory solver that identifies a feasible solution that satisfies the conjunct of constraints. If this is not feasible, the solver picks another Boolean assignment factoring in the learned conflict. In some sense, this can be viewed as a two level search problem where Boolean backbone identifies a *logical neighborhood* for the theory solver to search in.

Even though SMT is designed to efficiently explore disjunctive logic structures exploiting structure and symmetry in the encoding, the satisfiability task only requires reasoning about feasibility. In contrast, an OMT solver must search for other solutions once after identifying a feasible solution potentially with differing Boolean backbone.

In the notation as introduced for SMT, we solve problems of minimizing cost, $C(\vec{x})$ such that the CNF formula holds, where in the theory of LIA variables are constrained to be integral and $C(\vec{x})$ a linear function with integral coefficients.

OMT raises the specific challenges of both explicitly reasoning about disjunctions and optimizing a cost function. Particularly, disjunctions lead to local optimas which may not be connected since there are no convexity guarantees.

3.2 Integer Linear Programming (ILP)

A Integer Linear Programming (ILP) problem is parameterized by the tuple ($\mathbf{A}$, $\vec{b}$, $\vec{c}$). The objective is to solve for an optimal choice of $\vec{x}$ such that $\vec{c} \cdot \vec{x}$ is minimized. However, $\vec{x}$ is constrained to satisfy $\mathbf{A}\vec{x} \leq \vec{b}$, where $\leq$ represents element wise less than or equal to. Further, these vectors can be over a mixture of real numbers and integers. We note that strict inequalities $\vec{a_i} \cdot \vec{x} < b_i$ can be encoded as $\vec{a_i} \cdot \vec{x} \leq b_i - 1$.

Although MILP does not explicitly support disjunctions, Big M encoding can allow practitioners to implicitly approximate disjunctions by adding an additional decision variable. Disjunctions of inequalities that appear in the original encoding, $(\texttt{LHS}_1 \leq \texttt{RHS}_1) \vee (\texttt{LHS}_2 \leq \texttt{RHS}_2)$ can be encoded by adding the decision binary variable α. Since MILP must be expressed as a list of constraints that always hold, a large constant M is then added to the inequality to trivialize the constraints when α deselects the literal. For our example, the disjunction becomes encoded as the following two constraints:

$$\texttt{LHS}_1 \leq \texttt{RHS}_1 + \alpha M$$
$$\texttt{LHS}_2 \leq \texttt{RHS}_2 + (1 - \alpha) M$$

By using this encoding strategy, the assignment of α results in the selection of which clause must hold.

3.3 Graph Neural Network (GNN)

Recent progress in machine learning for optimization problems have been enabled by graph neural networks. In this section, we provide a brief introduction and key intuition behind these models. Interested readers can refer to (Wu et al. 2021b) for more details.

Graph Neural Networks (GNNs) are deep neural networks that take graph structured inputs and make predictions on both individual nodes or edges and the entire graph. As formalized by (Gilmer et al. 2017), these models output a graph or node representation (i.e., a high-dimensional vector), via message passing over the graph structure. The GNN takes node features x_v and applies i rounds of message passing where the hidden state h_v^i of each node is updated based a

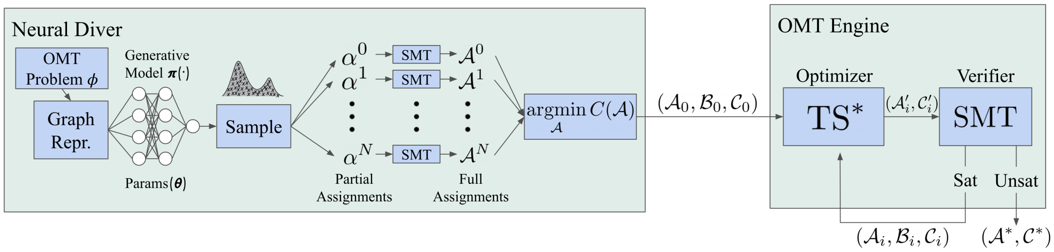

Figure 1: Ashera workflow. The neural diver serves as a warm-starter providing an initial low cost feasible solution. After neural diving, the OMT engine in Ashera alternates between a *Optimization-aware Theory Solver* (TS*) for optimization and an SMT solver for verification. Each invocation of TS* returns a tighter blocking clause, restricting the SMT solver to search for strictly lower cost solutions than the current model.

learned function parameterized by θ on it's neighbors' hidden states and the edge features: $m_v^{t+1} f_\theta(h_w^i, e_{vw})$ where w is a neighbor in the neighborhood of v, $N(v)$. The new hidden state is then $h_v^{i_1} = Agg(h_v^i, \sum_{w \in N(v)} m_v^{t+1})$.

Inspired by Convolutional Neural Networks (CNNs), which exploit the inductive bias that neighboring pixels are often related, Graph Convolutional Networks, or GCN (Kipf and Welling 2016), employ the same learned function across the same layer of the neural network irrespective of nodes. Analogous to computer vision, the shared function encourages the network to learn to recognize the same pattern occuring in connected subgraphs. This approach has seen wide success from analyzing social media graphs (Fan et al. 2022) to predicting molecule properties (Gilmer et al. 2017) and within combinatorial optimization has been used for network planning (Zhu et al. 2021) and chip placement (Mirhoseini et al. 2020).

Graph encoding for constrained programming. As done in (Gasse et al. 2019) and Nair et al. (Nair et al. 2020), one common graph representation for constrained programming is a graph, in which nodes represent constraints and variables. Edge between constraint nodes and variable nodes encode the coefficient of variables that appear in the constraint. Then, the message passing from variable to constraint in the graph neural network can be interpreted as how the variable embedding influence the constraint embedding (and vice versa). In practice, 2-3 rounds of massage passing suffice to model higher-order constraints, e.g., two variables in the same constraint, etc.

4 Method Overview

In this section, we provide an overview of our method, illustrated in Fig.1. Ashera consists of two main components: an OMT engine (Section 5) and a neural diver (Section 6).

The neural diver generates an initial feasible assignments $\mathcal{A}_0$ by first using a neural heuristic trained on solutions of similar OMT problems. Given a $\mathcal{A}_0$, the assignment uniquely specifies a cost $\mathcal{C}_0$ and a boolean backbone $\mathcal{B}_0$. The initial feasible solution provides a warm start to the OMT engine. We elaborate on how the neural diver is trained and how it selects initial feasible assignment in Section 6.

Given the tuple $(\mathcal{A}_0, \mathcal{B}_0, \mathcal{C}_0)$, the OMT engine generates a blocking clause: $C(x) < \mathcal{C}_0$. This restricts the engine to only search for lower cost assignments. Using the boolean backbone, our OMT engine iterates between an optimizer that searches for a lower cost solution than the current assignment $\mathcal{A}_i$, and a verifier that checks if the optimized assignment $\mathcal{A}'_i$ is optimal. In a counterexample-guided fashion, the optimizer utilizes feasible solutions returned by the verifier to refine the search. We elaborate on the optimizer and verifier in Section 5. When the verifier returns unsat, Ashera returns the best assignment, $\mathcal{A}^*$, and best cost, $\mathcal{C}^*$.

Ashera can be run in a *cold-start* settings when solutions to similar OMT problems are not available, or as a way to solve related OMT problems that are then used for training. In this setting, the neural diver can be replaced by a SMT solver. Although Ashera will not be able to learn from past examples, the SMT solver will still provide a valid albeit likely high cost assignment to the OMT engine.

5 OMT Engine

For exposition purposes, we first detail the cold-start setting of our OMT algorithm in this section, in which the neural diver is substituted with an SMT solver. In Section 6, we introduce the full Ashera algorithm, including the neural diver and introduce Logical Neighborhood Search.

5.1 Logical Neighborhood Search

Efforts like Wu et al. (Wu et al. 2021a) approach optimization problems as a large neighborhood search problem where the optimizer first discovers feasible solutions. Then, it explores assignments which differ from the feasible assignment by a ϵ-ball. Our approach identifies a more natural notion of logical locality for OMT.

After obtaining the tuple $(\mathcal{A}_i, \mathcal{B}_i, \mathcal{C}_i)$ from the SMT solver, we perform optimization with an optimality-aware theory solver TS* which takes in a Boolean backbone $\mathcal{B}$ as input. Unlike existing approaches to neighborhood search, we aim to search for solutions that have similar logic structure (i.e. preserves the same Boolean literal backbone). This definition of locality allows us to use an off-the-shelf ILP solver as the optimality-aware theory solver to conduct the neighborhood search.

Algorithm 1: **Ashera: OMT Solver**

Input: ϕ: OMT formula
Returns: $\mathcal{A}^*, \mathcal{C}^*$: best assignment and cost w.r.t. objective

$\mathcal{A}^*, \mathcal{C}^* := \emptyset, \infty$
$partialAssignSamples := neuralDiver(\phi)$
$SMT\_problem := createSMT(\phi)$
for all α **in** $partialAssignSamples$ **do**
    $isSat, (\mathcal{A}, \mathcal{B}, \mathcal{C}) := solve(SMT\_problem, \alpha)$
    if $\mathcal{C} < \mathcal{C}^*$ **and** $isSat$ **then**
        $\mathcal{A}^*, \mathcal{C}^* := \mathcal{A}, \mathcal{C}$
while True **do**
    $blocker := (cost \leq \mathcal{C}^* - \delta)$
    $isSat, \mathcal{A}, \mathcal{B}, \mathcal{C} := solve(SMT\_problem, blocker)$
    if not $isSat$ **then**
        break
    $ILP\_problem = createILP(\mathcal{B})$
    $\mathcal{A}^*, \mathcal{C}^* := solve(ILP\_problem)$
return $\mathcal{A}^*, \mathcal{C}^*$

For each literal in the Boolean backbone $\mathcal{B}_i$, we add the negation of the false literal as an ILP constraint and the literal itself for true literals in the backbone. In this way, we do not need to encode disjunctions into the ILP encoding using Big M or convex hull. In section A.5, we discuss some sound optimizations that can be done with this constraint generation process. By restricting to the convex region around the feasible solution, we can express the optimization purely as the conjunction of literals in $\mathcal{B}_i$. As we have identified a feasible solution $\mathcal{A}_i$, we effectively use ILP to improve on the found solution within the neighborhood which maintains the same logic assignment. Note that the search is localized to a connected region specified by the constraints, in which at least one feasible solution can be found (i.e., the current solution $\mathcal{A}_i$). However, unlike a LIA theory solver, ILP is cost-aware and able to optimize with respect to our objective function. The ILP solver produces a tuple $(\mathcal{A}'_i, \mathcal{C}'_i)$ as output, where $\mathcal{A}'_i$ is an assignment that achieves the optimized cost $\mathcal{C}'_i$ within the neighborhood.

To reach another disconnected region, we query SMT with the optimized cost $\mathcal{C}'_i$ for a feasible solution that's strictly better than the solution $\mathcal{A}'_i$ discovered by the last iteration of ILP. If this results in an unsatisfiable result, we terminate knowing there exists no better feasible solution to the OMT problem. Given ILP discovered the optimal solution for the particular logic backbone, the SMT solver will find a feasible solution with a different logic backbone.

We present our algorithm in full in Algorithm 1 noting that a tolerance, δ, can be set depending on the user's domain expertise. For our integer example, $\delta = 1$ is natural.

6 Neural Diving

Inspired by work by Nair et al. (Nair et al. 2020), we adopt a neural diver which can be thought of as a warm-starter that identifies promising initial feasible solutions.

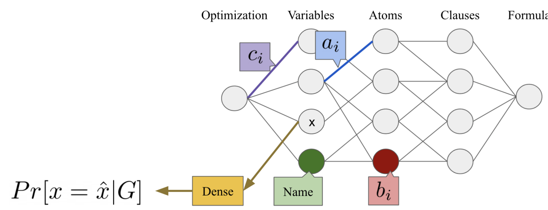

Figure 2: Graph representation. We represent the OMT problem as a graph with edges connecting variable nodes and constraint/cost nodes. The logic structure is represented similarly to an abstract syntax tree (AST), with the AST leaf nodes coinciding with the atomic constraint nodes.

6.1 Graph Representation

We translate an OMT problem into a graph by encoding each variable as a node; we encode the variables and atomic formula as is done in (Gasse et al. 2019). We encode the elements of $\vec{a_{ij}}$ as weights on edges between variables and constraints. We add $\vec{b_{ij}}$ as node attributes for constraint nodes and variable names as node attributes on variable nodes. We also have a cost node connected to each of the variable nodes with weights corresponding to $\vec{c}$. Each atomic formula node serves as leaves in an adjoining tree representing the clauses and final formula.

Taking from the approach in (Nair et al. 2020), we implement a neural partial assignment generator based on a graph convolutional neural network (GCN). As shown in Fig. 2, we build a graph connecting variable nodes and literal/constraint nodes with the edge weights indicating the linear coefficients in the literal. Finally, we encode the cost objective as special literal node, with the coefficients as edge weights.

In contrast to the encoding for Nair et al. (Nair et al. 2020), OMT also includes disjunctions over the literals. As such, we encode the disjunction as a tree over the literal nodes. Disjunction and conjunction nodes pass information between related clauses in rounds of message passing.

6.2 Formulating the Learning Problem

We consider the setting where an OMT solver is repeatedly solving similar problems. This means we can curate a training set of problems in this distribution based on historical queries or in simulation. Further, by design, our OMT engine can be run without the neural diver by replacing it with an initial SMT call. The cold-start OMT engine can be used to label training examples with optimal models. Using this labeled training set, we seek to reduce the required time to solve unseen problems from the same distribution.

With the graph encoding, call it G, our goal is to learn a function, f that estimates for each variable a probability distribution over potential values. We do this with a standard graph convolutional network (GCN) (Kipf and Welling 2016). We learn this function f over examples G_i labeled with x_i^*, a cost optimal variable assignment. We treat integer variable values as independent classes train the model to classify each variable.

We use a GCN to learn an embedding for each variable, which we then pass through a linear layer to predict the class corresponding to the variable assignment. We find it sufficient to run two rounds of message passing for this application. With two rounds, the variables nodes can be aggregate information from two hop neighbors allowing the final learned embedding of the variable to be both influenced by variables that it shares an atomic constraint with and the clause that it belongs to.

Using the learned embedding, we optimize the following cross entropy loss: $\mathcal{L} = -\sum_{i=1}^{m} x_i^* \log p(x_i|G)$ where x_i^* is the optimal assignment of the i-th variable and $\log p(x_i|G)$ is the probability of the assignment generated by the function f.

This loss intuitively maximizes the probability that the optimal assignment is selected. Note for each variable the GCN effectively approximate the probability distribution over variable assignments conditioned on G, $p(x|G)$. This is the desired f we sought to learn.

6.3 Partial Assignment Warm-Start

When the neural diver is used to solve a problem of interest, the diver makes a prediction based on the input problem G. This inference results in an estimated probability $P(x|G)$ returned as output logits. We sample variable assignments based on these logits and use the KL divergence to a unifrom distribution to estimate confidence. If the KL divergence is larger than a confidence threshold, C, we abstain from assigning (see Appendix A.7 for implementation details). This results in partial assignments α^i as only variables that are easy to predict have assignments. To get full feasible solutions $\mathcal{A}^i$, we call an SMT solver on each partial assignment to get a complete feasible assignment. We do this by adding to the existing OMT formula equality constraints $x = k$ where x is a variable and k is the sampled assignment from the partial assignment generator.

We then use Ray (Moritz et al. 2018) to search for a valid assignment in parallel for a user specified time, T, with K parallel threads. If a thread discovers an unsatisfiable partial assignment, it continues searching with another sample from the generative model until the time expires. After running for T, the diver returns the best assignment discovered. In the case that it does not find any feasible solutions, the OMT engine runs SMT first to get a feasible assignment. By default Ashera uses $T = 5s$, $K = 5$, and $C = 1$, but these parameters can be tuned on a validation set in practice for each problem family.

7 Experiment Setup

For this work, we look at two families of real-world OMT problems: 1) DAG job scheduling, and 2) multi-agent traveling salesman problem (TSP). Disjunctions native to these two families are particularly challenging as they result in a disconnected feasible set and a redundancy of equally optimal solutions. For instance, in DAG scheduling, the each assignment of tasks to resources defines a nontrivial scheduling subproblem, that tend to be similar but not identical to each other. We describe the dataset generation, baseline solvers in this section, and include experimental details such as hardware setup in Appendix A.6.

Dataset Generation. In order to train and evaluate a learning-based OMT solver, we generate instances for each family of problems labeled with their optimal assignment.

Baseline Solvers. To compare with existing ILP solvers (e.g., Gurobi), we encode the OMT benchmarks as ILP problems using the standard Big M encoding to approximate the disjunctions. Big M encoding introduces an additional binary choice variable α to determine which clause in the disjunction is enforced as an ILP constraint. This increases the number of variables combinatorially in the number of disjunctions and and requires the ILP solver to optimize over all disjunctive branch simultaneously.

For a fair comparison, the models were evaluated without GPU assistance, but we expect improved performance if accelerators are available at inference time. As Gurobi is highly parallelized, to make fair comparison of algorithmic cost, we compare results based on process time unless otherwise noted and impose a 1 hr limit for experiments. We do not include training time in our comparison as it required only 2 hour on a single GPU and depends heavily on the amount of data. Further, targeted applications of repeated OMT solves occur on a weekly basis allowing for training between invocations.

8 Results

In this section, we present our empirical results of existing OMT tools, *vZ*, OptiMathSAT, Gurobi (ILP with Big M), and Ashera. Our results show that Ashera scales to larger problems, outperforming all three baselines by as much as 5x compared to the next best solver on the scheduling and multi-agent TSP tasks. We compare performances with Par-2 score, which is the solving time for solved instances and two times timeout for unsolved ones.

8.1 Task Scheduling for Directed Acyclic Graphs

We generate a total of 43,596 similar DAG scheduling problems using the same problem encoding but varying the number of tasks and CPUs. We split up the evaluation based on the number of tasks in the scheduling problem. For a realistic setting, we consider two GPUs, and we have all the task with the same expected runtime of 15 seconds and release times of 2. This symmetry is notoriously difficult for traditional ILP solutions. To further make the instances comparable and always feasible, we scale up the deadline as the number of tasks increase, and only have one randomly placed dependency between two tasks in the taskset. All task within a problem instance have the same deadline as reported in Table 4. In this section, we only consider Ashera trained and tested on the *same* number of tasks and defer analysis Ashera's ability to adapt to tasks sizes not seen in training in Section 8.3. We use $T = 1m$ for 10 tasks and default values otherwise.

We report the performance of baseline solvers and Ashera in Table 1 categorized by the number of tasks in the problem instance. The table indicates that existing baselines OptiMathSAT has the best performance until 8 tasks while *vZ*

Table 1: OMT Solver Performance on Scheduling. We report PAR-2 scores of process time to account for timeouts and provide number of solved problems in parenthesis.

Number of Tasks	Average PAR-2 Score in Seconds (Number Solved in 1 hr) Gurobi	*vZ*	OptiMathSAT	Ashera	Ashera Cold-start
5	2.21 (20)	**0.02** (20)	**0.02** (20)	2.25 (20)	2.35 (20)
6	2.30 (20)	0.11 (20)	**0.03** (20)	2.31 (20)	2.40 (20)
7	2.78 (20)	1.00 (20)	**0.17** (20)	2.33 (20)	2.50 (20)
8	3.84 (20)	16.27 (20)	**1.28** (20)	3.04 (20)	3.44 (20)
9	26.29 (20)	149.39 (20)	31.17 (20)	16.78 (20)	**13.95** (20)
10	400.01 (20)	5246.23 (9)	841.21 (20)	**135.97** (20)	181.34 (20)
11	3562.15 (15)	7200.00 (0)	7200.00 (0)	3204.00 (18)	**2924.77** (18)

performs badly on 10 tasks. Gurobi however, continues to solve instances with 11 tasks where *vZ* and OptiMathSAT timeout. Ashera in contrast is 3x faster and solves three more problems than Gurobi on problems with 10 and 11 tasks, respectively. The results demonstrates the effectiveness of Ashera when the problem size scales. This suggests the benefit of applying Ashera to large-scale real-world applications, which typically have hundreds of variables.

8.2 Multi-agent TSP

In our multi-agent TSP benchmark, we generate 2500 instances of TSP with two clusters of waypoints arranged in a polygon, and additionally 22 problems for testing. We vary the distance between the center of the cluster and the origin where the vehicles start and vary the radius of the polygon. For simplicity, we provide as many vehicles as there are clusters and ensure restrictions on weight are not constraining. Additionally, the first waypoint is always the starting point for the vehicles.

Table 2 shows Ashera outperforms all three baselines on the largest problems, solving faster than OptiMathSAT while Gurobi and *vZ* solve none. In Table 2, we report Ashera's performance when trained on instances with the same number of waypoints as those of the test set. We use $C = 0.5$ and otherwise use default parameters.

8.3 Cold-Start Ashera

We ablate the learned component of Ashera and see that Ashera performs faster on scheduling problems with 5 to 8 tasks. We attribute this to a 2 second overhead incurred in order to perform neural network inference on CPU and initialize Ray (Moritz et al. 2018). We leave for future work improvements afforded by hardware accelerators such as GPUs. Cold-start Ashera outperforms neural Ashera on scheduling problems with 11 tasks. For multi-agent TSP, Table 2 shows neural diving provides modest improvement compared to cold-start due to a larger number of variables and constraints. We note that Cold-start Ashera already outperforms or matches performances of the baselines.

8.4 Neural Diver Performance

One important aspect for neural-based solvers is its performance to transfer to similar but new problems. Specifically,

[0]We generated scheduling benchmarks of 11 to 20 tasks but all methods timeout with 1 hour at 11 tasks. Our benchmark contains multi-agent TSP problems of 8 to 10 waypoints per cluster but all methods timeout with 1 hour at 8 waypoints per cluster.

Table 2: OMT Solver Performance on Multi-Agent TSP. We report PAR-2 scores of process time to account for timeouts and provide number of solved problems in parenthesis.

# Waypoints per Cluster	Average PAR-2 Score in Seconds (Number Solved in 1 hr) Gurobi	*vZ*	OptiMathSAT	Ashera	Ashera Cold-start
3	0.91 (22)	0.16 (22)	**0.12** (22)	2.48(22)	3.93 (22)
4	76.51 (22)	**0.80** (22)	0.94 (22)	4.51(22)	7.89 (22)
5	5154.00 (10)	**5.74** (22)	7.85 (22)	20.50(22)	24.29 (22)
6	7200.00 (0)	7200.00 (0)	**90.54** (22)	104.00(22)	102.91 (22)
7	7200.00 (0)	7200.00 (0)	919.26 (22)	**754.33** (22)	793.88 (22)

Table 3: Ablation on Neural Diving. We both test performance transfer to larger problems trained on smaller problems (Curriculum) and applying neural diving on existing solvers (Neural Diver + Gurobi/OptiMathSAT).

Number of Tasks	Average PAR-2 score in Seconds (Number Solved in 1 hr) Neural Diver + Gurobi	Neural Diver + OptiMathSAT	Ashera	Ashera Curriculum
5	2.34 (20)	**2.37** (20)	2.25 (20)	-[1]
6	2.42 (20)	**2.43** (20)	2.31 (20)	2.28 (20)
7	2.66 (20)	**2.44** (20)	2.33 (20)	2.30 (20)
8	4.00 (20)	**3.24** (20)	3.04 (20)	2.36 (20)
9	19.44 (20)	12.03 (20)	16.78 (20)	**13.97** (20)
10	288.09 (20)	288.09 (20)	**135.97** (20)	174.19 (20)
11	**2649.02** (17)	5728.87 (9)	3204.00 (**18**)	3118.62(17)

we look at two settings: 1) the performance of Ashera when tested on larger problems than seen in training, and 2) the neural diver applied to Gurobi and OptiMathSAT.

Performance transfer to larger problems. In this setting, Ashera is trained only on problems that are smaller than the test-time number of task. On scheduling, Table 3 shows Ashera performs comparably to when it is trained on the same sized problems. On TSP, the performance is also comparable (see Appendix). This ablation indicates that Ashera can be trained on smaller problems from the same family to scale to larger problems of interest.

Neural Gurobi and Neural OptiMathSAT. We further consider the performance of the learned neural diver on Gurobi and OptiMathSAT. For this setting, we replace the Ashera OMT engine with Gurobi or OptiMathSAT. As the diver provides a Z3 verified upper bound on the cost, all OMT solvers are compatible and can use the upper bound as a hint. On problems with 11 tasks, OptiMathSAT can solve 9 problems compared to none before and Gurobi can solve two additional problems. This demonstrates the use of neural guidance on OMT problems extends beyond our specific solver design.

9 Conclusions

Our work presents Ashera, a neural OMT solver, which performs up to 3x faster and solves three more problems than the widely-used commercial solver, Gurobi, on problems with 10 and 11 tasks, respectively. Traditional solvers, OptiMathSAT and Z3, are unable to solve any problems of 11 tasks within 1 hour timeout. Further, Ashera is 18% faster than OptiMathSAT on multiagent TSP with 15 waypoints, where Gurobi and Z3 timeout. As OMT problems solved in

[1]We do not evaluate on 5 tasks as there are no smaller problems.

practice tend be solved on a regular basis, we make the case for learned-based OMT solver that train on a set of similar problems encountered previously in the application or in simulation. We contribute benchmark of problem families including DAG task scheduling and multi-agent TSP for evaluating learning-based OMT solvers.

10 Acknowledgements

This research is supported in part by NSF CISE Expeditions Award CCF-1730628, NSF NRI #2024675 and under the NSF AI4OPT Center. UC Berkeley research is also supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Meta, Futurewei, Google, Intel, Microsoft, Nvidia, Sco tiabank, Splunk and VMware.

References

Balcan, M.; Dick, T.; Sandholm, T.; and Vitercik, E. 2018. Learning to Branch. *CoRR*, abs/1803.10150.

Barrett, C.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 2009. Satisfiability Modulo Theories. In Biere, A.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, chapter 26, 825–885. IOS Press.

Beauchemin, M. 2014. Apache Airflow.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Bertolissi, C.; Dos Santos, D. R.; and Ranise, S. 2018. Solving multi-objective workflow satisfiability problems with optimization modulo theories techniques. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, 117–128.

Bjørner, N.; Phan, A.-D.; and Fleckenstein, L. 2015. νZ - An Optimizing SMT Solver. In Baier, C.; and Tinelli, C., eds., *Tools and Algorithms for the Construction and Analysis of Systems*, 194–199. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-662-46681-0.

De Moura, L.; and Bjørner, N. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

Dutertre, B.; and De Moura, L. 2006. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2): 1–2.

Fan, W.; Ma, Y.; Li, Q.; Wang, J.; Cai, G.; Tang, J.; and Yin, D. 2022. A Graph Neural Network Framework for Social Recommendations. *IEEE Transactions on Knowledge and Data Engineering*, 34(5): 2033–2047.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*, 1263–1272. PMLR.

Gog, I.; Kalra, S.; Schafhalter, P.; Gonzalez, J. E.; and Stoica, I. 2022. D3: A Dynamic Deadline-Driven Approach for Building Autonomous Vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 453–471. Association for Computing Machinery.

Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual.

Henry, J.; Asavoae, M.; Monniaux, D.; and Maïza, C. 2014. How to Compute Worst-Case Execution Time by Optimization modulo Theory and a Clever Encoding of Program Semantics. LCTES '14, 43–52. Association for Computing Machinery.

Hopfield, J. J.; and Tank, D. W. 2004. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52: 141–152.

Katz, G.; Barrett, C.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International conference on computer aided verification*, 97–117. Springer.

Kipf, T. N.; and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Kovásznai, G.; Biró, C.; and Erdélyi, B. 2017. Generating Optimal Scheduling for Wireless Sensor Networks by Using Optimization Modulo Theories Solvers. In *SMT*, 15–27.

Leino, K. R. M. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, 348–370. Springer.

Leofante, F.; Ábrahám, E.; Niemueller, T.; Lakemeyer, G.; and Tacchella, A. 2017. On the synthesis of guaranteed-quality plans for robot fleets in logistics scenarios via optimization modulo theories. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, 403–410. IEEE.

Manolios, P.; Pais, J.; and Papavasileiou, V. 2015. The Inez Mathematical Programming Modulo Theories Framework. In Kroening, D.; and Păsăreanu, C. S., eds., *Computer Aided Verification*, 53–69. Springer International Publishing.

Mirhoseini, A.; Goldie, A.; Yazgan, M.; Jiang, J.; Songhori, E.; Wang, S.; Lee, Y.-J.; Johnson, E.; Pathak, O.; Bae, S.; et al. 2020. Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*.

Moritz, P.; Nishihara, R.; Wang, S.; Tumanov, A.; Liaw, R.; Liang, E.; Elibol, M.; Yang, Z.; Paul, W.; Jordan, M. I.; and Stoica, I. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 561–577.

Moura, L. d.; and Bjørner, N. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

Nair, V.; Bartunov, S.; Gimeno, F.; von Glehn, I.; Lichocki, P.; Lobov, I.; O'Donoghue, B.; Sonnerat, N.; Tjandraatmadja, C.; Wang, P.; et al. 2020. Solving Mixed Integer Programs Using Neural Networks. *arXiv preprint arXiv:2012.13349*.

Sebastiani, R.; and Trentin, P. 2015. OptiMathSAT: A Tool for Optimization Modulo Theories. In Kroening, D.; and Păsăreanu, C. S., eds., *Computer Aided Verification*, 447–454. Springer International Publishing.

Selsam, D.; Lamm, M.; Bünz, B.; Liang, P.; de Moura, L.; and Dill, D. L. 2019. Learning a SAT Solver from Single-Bit Supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Wu, Y.; Song, W.; Cao, Z.; and Zhang, J. 2021a. Learning Large Neighborhood Search Policy for Integer Programming. In *Advances in Neural Information Processing Systems*.

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Yu, P. S. 2021b. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1): 4–24.

Yang, Z.; Chiang, W.-L.; Luan, S.; Mittal, G.; Luo, M.; and Stoica, I. 2022. Balsa: Learning a query optimizer without expert demonstrations. *arXiv preprint arXiv:2201.01441*.

Zhu, H.; Gupta, V.; Ahuja, S. S.; Tian, Y.; Zhang, Y.; and Jin, X. 2021. Network Planning with Deep Reinforcement Learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 258–271. Association for Computing Machinery.

Appendix A

A.1 DAG Scheduling with Deadlines

The scheduling problem belongs to a widely applicable family of scheduling problems. It requires discovering the optimal placement of tasks to resources as well as assigning start times to tasks. Further, assignments must satisfy constraints on both resources and dependencies between tasks. We desire to maximize slack – the buffer time before the deadline that a task is expected to complete. As such it's often nontrivial to find any feasible schedule, much less an optimal one.

This family of problems appears in both the workflow management platform Apache Airflow (Beauchemin 2014) and in the DAG scheduler used for the dynamic deadline-driven execution model for self driving (Gog et al. 2022).

We consider a set of N tasks, $T = \{t_i | i \in [1, N]\}$, and a dependency matrix M where $M_{ij} = 1$ if t_i must complete before t_j otherwise 0. We further consider a set of deadlines and expected runtimes denoted as d_i and e_i, respectively.

In addition to runtime, we also consider resource requirements and placements. We denote r_i to be 1 if t_i requires a GPU and 0 otherwise.

We seek to optimize with respect to two sets of variables s_i and p_i which denote the start time and placement of t_i. Let N_G and N_C be the number of GPUs and CPUs, respectively. We encode $p_i = k$ to be in $[1, N_G]$ if it's placed on the k^{th} GPUs of N_G and $(k - N_G)^{th}$ CPU if it's in $[N_G, N_G + N_C]$.

Our cost objective is

$$\sum_{i=0}^{N} d_i - (s_i + e_i)$$

with the following constraints:

- **Basic constraints.** For all i, $0 \leq s_i$ and $0 < p_i$.
- **Finish before deadline.** For all i, $s_i + e_i \leq d_i$.
- **Placement constraints.** For all i, if $r_i = 1$, $1 \leq p_i \leq N_G$. Otherwise, $r_i = 0$, $1 \leq p_i \leq N_G + N_C$.
- **Dependency Respecting.** For all i, j, if $M_{ij} = 1$, $s_i + e_i \leq s_j$.
- **Exclusion.** For all i, j, $p_i = p_j \implies (s_i + e_i \leq s_j \vee s_j + e_j \leq s_i)$.

Big M for Scheduling. Unfortunately, the exclusion constraint requires a disjunction. In order to compare against ILP solvers, we use the Big M strategy as presented in 2. We introduce an additional two variables per pair of tasks i, j to 1) choose if task i and task j utilize the same resources and 2) choose if task i completes execution before task j begins or vice versa. We provide the Big M version of the exclusion constrain in Appendix A.3.

A.2 Multi-Agent Traveling Salesman Problem

The multi-agent Traveling Salesman Problem (TSP) appears in practical route planning applications including package deliveries in warehouse operations. A multi-agent TSP is specified by the distances between the W waypoints and the number of vehicles V. In this problem, the optimizer must find an ordering o_i in which waypoint, i, is visited by vehicle, v_i. We denote the starting waypoint as s. Our objective is to minimize the sum of the times t_i when a waypoint is visited:

$$\sum_{i=0}^{N} t_i$$

Due to space constraints, we highlight the following constraints and present the full encoding for multi-agent TSP in Appendix A.4:

- **Visited.** All waypoints w must be visited by at least one vehicle.
- **Deterministic.** After visiting a waypoint, w, a vehicle visits at most one waypoint w' immediately afterwards.
- **Ordering.** The starting waypoint has $o_s = 0$. For all waypoints, w, visited in order, o_w, the waypoint's predecessor p_w must have order $o_{p_w} = o_w - 1$. This prevents tours that do not include the starting point.
- **Weight Constraint.** The sum of the weights of the vehicles is less than a given value M.
- **Visit Time.** For all waypoints w, the visit time t_w if vehicle v_w visits it is at least the $t_{p_w} + \tau_{v,p,w}$ where t_p is the time when the preceding waypoint was visited and τ is the travel time from p to w by vehicle v.
- **Exclusion.** If vehicle v is traveling from w to w' from t_w to $t_{w'}$, there cannot be a waypoint w'' visited by v while it's traveling.

A.3 Big M for multi-agent TSP.

We again use the Big M encoding to encode disjunctions. The most complex disjunction requiring the disjunction of conjunctions is the ordering condition for a waypoint w:

$$\bigvee_{w' \in W, v \in V} \mathbf{M}_{v,w',w} \wedge (o_{w',v} = o_{w,v} - 1)$$

In the disjunction of conjuncts, all the inequalities in the conjuct share the same choice variable, ensuring that all constraints in the disjunctive case hold simultaneously if chosen.

Big M Exclusion Constraint We use Big M to encode the exclusion constraint for MILP. The implication

$$p_i = p_j \implies (s_i + e_i \leq s_j \vee s_j + e_j \leq s_i)$$

can be encoded as

$$(p_i < p_j) \vee (p_j < p_i) \vee (s_i + e_i \leq s_j) \vee (s_j + e_j \leq s_i).$$

By creating two binary variables $\alpha_{i,j}$ and $\beta_{i,j}$ per constraint, we introduce the following constraints in replacement using Big M:

- **Case 1** $(\alpha_{i,j} = 0, \beta_{i,j} = 0)$ **:**
$$p_i - p_j < M\alpha_{i,j} + M\beta_{i,j}$$
- **Case 2** $(\alpha_{i,j} = 0, \beta_{i,j} = 1)$ **:**
$$p_j - p_i < M\alpha_{i,j} + M(1 - \beta_{i,j})$$
- **Case 3** $(\alpha_{i,j} = 1, \beta_{i,j} = 0)$ **:**
$$s_i + e_i - s_j \leq M(1 - \alpha_{i,j}) + M\beta_{i,j}$$
- **Case 4** $(\alpha_{i,j} = 1, \beta_{i,j} = 0)$ **:**
$$s_j + e_j - s_i \leq M(1 - \alpha_{i,j}) + M(1 - \beta_{i,j})$$

A.4 Multi-Agent Traveling Salesman Problem Full Encoding

The multi-agent traveling salesman problem can be defined as an optimization problem where the aggregation over the time when the waypoints are visited is minimized.

We index the vehicles and waypoints respectively from 0 to $|V|-1$ and $|W|-1$, where V and W are the set of vehicles and waypoints and $|\cdot|$ denotes the cardinality of a set. The constraints are defined over the following variables:

- u_v - whether or not a vehicle is being used
- $\mathbf{M}_{v,w,w'}$ - a Boolean 3D array indicating if vehicle v travels from waypoint w to waypoint w'
- p_w - the preceding waypoint from which the vehicle visits w
- x_w - the vehicle that visits waypoint w
- h - starting waypoint
- o_w - order that waypoint w is visited by a vehicle v. Note that this is an ordering per vehicle not globally for all vehicles.
- t_w - the time when waypoint w is visited
- m_{max} - total mass allowed

We get the following constants from an oracle.

- $\tau_{v,w,w'}$ - time for agent v to travel from w to w'
- $c_{v,w,w'}$ - energy consumption
- γ_v - vehicle weight.

Our optimization seeks to minimize the aggregated time t when the waypoints are visited under the following constraints:

1. Each waypoint except the harbor must be visited by a vehicle:
$$\forall w' \in W \setminus \{h\}. \sum_{v\in V, w\in W} \mathbf{M}_{v,w,w'} = 1$$
The harbor has to be visited by the vehicles that are used.
$$\forall v \in V. \sum_{w\in W} \mathbf{M}_{v,w,h} = u_v$$
2. From one waypoint only one other waypoint is visited next (determinism), and according to fixed order:
$$\forall w \in W \setminus \{h\}. \sum_{v\in V, w'\in W\setminus\{h\}} \mathbf{M}_{v,w,w'} = 1$$
Vehicles that are used should leave the harbor.
$$\forall v \in V. \sum_{w'\in W} \mathbf{M}_{v,h,w'} = u_v$$
3. No self loop allowed.
$$\forall v \in V, w \in W. \overline{\mathbf{M}_{v,w,w}}$$
4. If a point has order o then it must have been reach from another point with order $o-1$. For the harbor starting point we have,
$$\forall v \in V. o_{h,v} = 0$$
for $o_{w,v}$ where v is unused we similarly constrain it to be zero:
$$\forall w \in W \setminus \{h\}, v \in V. \\ (o_{w,v} = 0) \vee \bigvee_{w'\in W} \mathbf{M}_{v,w,w'}$$
and also
$$\forall w \in W \setminus \{h\}. \\ \bigvee_{w'\in W, v\in V} (\mathbf{M}_{v,w',w} \wedge o_{w',v} = o_{w,v} - 1)$$
5. For each waypoint w, a vehicle must visit another waypoint w' from w if it travels from some waypoint w'' to w:
$$\forall v \in V, w \in W. \\ (\sum_{w''\in W} \mathbf{M}_{v,w'',w} = 1 \rightarrow \sum_{w'\in W} \mathbf{M}_{v,w,w'} = 1)$$
6. Constraint for p_w:
$$\forall w \in W. p_w = \sum_{v\in V, w'\in W} w' \cdot \mathbf{M}_{v,w',w}$$
7. Constraint for x_w:
$$\forall w \in W. x_w = \sum_{v\in V, w'\in W} v \cdot \mathbf{M}_{v,w',w}$$
8. A vehicle is used when:
$$\forall v \in V.(u_v \leftrightarrow \bigvee_{w,w'\in W} \mathbf{M}_{v,w,w'})$$
9. The total weight is less than a given value:
$$\sum_{v\in V} u_v \cdot \gamma_v < m_{max}$$
10. The total time each agent takes is equal to t, which is in the minimization problem:
$$\sum_{v\in V, w\in W, w'\in W\setminus\{h\}} \mathbf{M}_{v,w,w'} \cdot \tau_{v,w,w'} = t$$

A.5 Literal Dropping

To further improve performance, we recognize that literals can be dropped safely if the formula is expressed as negation normal form (NNF), in which the negation operator is only applied to atoms and the only allowed Boolean operators are conjunction and disjunction. We formally state the observation as follows.

Observation 1. *Given a satisfying assignment $\mathcal{A}$ to an SMT formula ϕ in NNF over LIA, let L^+ and L^- be the sets of true and false (theory) literal that appear in ϕ by applying $\mathcal{A}$. Literals $l \in L^+$ define a valid solution space for variables in ϕ. That is, $l \in L^-$ can be dropped while maintaining soundness.*

Proof. First, observe that removing false literals $l \in L^-$ in an NNF ϕ does not affect the satisfiability of ϕ (for $l \in L^+$ alone satisfies ϕ). Thus, passing L^+ to an LIA theory solver, any solution returned by the theory solver must satisfy $l \in L^+$ and hence will satisfy ϕ. □

Moreover, since $L^+ \cup L^-$ imposes more constraints than L^+ does, the optimal value found in L^+ can be greater (resp. smaller) than that found in $L^+ \cup L^-$ when maximizing (resp. minimizing) an objective. This allows us to push bounds even further in the ILP solving phase.

Summary of Benchmarks Table 4 and Table 5 provide a summary of the number of variables and constraints for each size of problem in DAG scheduling and multiagent TSP, respectively.

Table 4: Scheduling Benchmark Summary.

Number of Tasks[2]	Number of Training Cases	Number of Test Cases	Average Variable Count	Average Constraint Count	Deadline
5	811	20	16	48	50
6	1459	20	19	63	57
7	2383	20	22	80	65
8	3630	20	25	99	72
9	5250	20	28	120	80
10	7291	20	31	143	87
11	9801	20	-	-	-
12	12831	20	-	-	-

Table 5: Multi-Agent TSP Benchmark Summary.

# Waypoints per Cluster	Number of Training Cases	Number of Test Cases	Average Variable Count	Average Constraint Count
3	2500	22	137	428
4	2500	22	211	654
5	2500	22	301	928
6	2500	22	401	1250
7	2500	22	519	1620
8	2500	22	-	-

A.6 Hardware Setup and Software Versions

For evaluation, we used c5.xlarge AWS cloud instances, which have 3.0 GHz Intel Xeon Platinum processors with 4 vCPUs and 8 GiB of RAM. For training, we used two Quadro GV100 GPUs with 32GB GPU Memory. We implemented parallel assignment search using Ray 1.10 (Moritz et al. 2018), a distributed programming framework. Our evaluation uses the following solver versions as baselines: Gurobi v9.5.1 (Gurobi Optimization, LLC 2021), Z3 4.8.16 (De Moura and Bjørner 2008), and OptiMathSAT 1.7.3 (Sebastiani and Trentin 2015). The same version of Gurobi is used for Logical Neighborhood Search and likewise of Z3 as the verifier in completing partial assignments and verifying soundness in the OMT engine.

A.7 Implementation details of diver.

KL divergence. We use the KL divergence to compare our learned distribution from a uniform distribution over the allowed variables values. We use static analysis to extract single variable inequalities ($k < x$) to obtain simple upper and lower bounds.

If the learned distribution is similar to this restricted uniform distribution it is likely the variable is symmetric in the data and an assignment to the variable is likely to precipitate simpler subproblems.

Parallel search. To augment the parallel search, we additionally ran a thread that simply ran the OMT engine in parallel with the diving exploration. This way if the problem is small and easy to solve, the solver will not be penalized to severely for diving.

The diver on the other hand explored up to 5 partial assignments at a time as sampled from the generative model. This pool of partial assignments always includes the highest probability partial assignment. This assignment is gotten by taking the highest probability class from generative model using an argmax. We found this practically useful as this assignment is an often promising partial assignment but as the number of predicted variables increases the odds of getting this particular sample decreases.