# Ashera: Neural Optimization Modulo Theory

Justin Wong*†, Pei-wei Chen*†, Tianjun Zhang†,
Joseph E. Gonzalez†, Yuandong Tian‡, Sanjit A. Seshia†
†University of California, Berkeley, ‡Meta AI Research

***Abstract*—Applications of 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. 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 5.7x speedup and shows improved scaling compared to MILP approximation as used in industry and state-of-the-art OMT solvers.**

## I. Introduction

The application of neural network to learn heuristics (i.e., *neural guidance*) for solving optimization problems has shown improved system performance, e.g., learned query optimization [1], chip placement [2], etc. However, unlike traditional solutions that leverage existing solvers, the neural network components in these systems are learned on the fly to solve specific combinatorial optimization problems. This can be improved if we exploit shared backbone strategies provided by heavily optimized solvers. Our work strives to take steps towards a unified framework for exploiting both solver techniques and neural guidance.

These combinatorial optimization problems can be expressed using Optimization Modulo Theory(OMT), which builds on Satisfiability Modulo Theory by requiring an objective function. Solutions to OMT need to be guaranteed to be not just feasible but optimal, which is strictly harder to find. Existing approaches either use the Big M approximation to reduce the problem to ILP [3] or do iterative calls to SMT [4], [5], which scales poorly. In both case, the solvers are designed to be general purpose and problem agnostic. Manual effort on tuning and expert knowledge are often required for addressing scalability issues. Such tuning can often improve the solving time from days to a matter of minutes.

However, the efficiency of existing solvers can be substantially improved since they fail to transfer knowledge of solving one problem instance to another similar one. We observe that optimizations in practice are done repeatedly, for example monthly for network planning. As such, the problems to be solved on a regular basis differ only slightly, but the optimization procedure needs to be redone, as modified constraints can change the solution. As a result, learning from previous solved problems can speed up the next optimization without any manual human input.

With this insight, we present a neural OMT solver Ashera which facilitates solving by drawing insights from previous solved problems. Ashera consists of a neural diver and an OMT engine. The OMT engine iterates between calling an SMT solver and an optimality-aware theory solver, TS* where the SMT solver finds strictly better solutions and verifies optimality while the theory solver solves for the optimal solution within the neighborhood of the solution given by the SMT solver. Trained on similar problems from the same problem family, the neural diver tries to predict promising partial assignments that lead to better cost when extended to full assignments. This approach exploits an often overlooked opportunity to practically improve performance potentially, reducing the manual expert tuning required to solve disjunctive optimization problems in practice.

As existing benchmarks provide dozens of diverse OMT problems, we build a benchmark for evaluating learning-based OMT solvers. Our benchmark provides over 35,000 OMT problems from two families of problems: task scheduling and multi-agent Traveling Salesman Problem (TSP). Using this benchmark we demonstrate 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:

- Present 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 LIA, but the framework generalizes to other theories.
- Ashera exhibits as much as 5.7x speed up compared to the widely-used commercial solver Gurobi and solves scheduling problems of a scale on which other OMT solvers *vZ* and OptiMathSAT timeout.

We organize the rest of the paper as follows. In Section II and III we introduce related works and preliminaries. An overview of our approach is then presented in Section IV and details are explained in Section V and VI. We present experiments and results in Section VII and VIII and lastly conclude in Section IX.

*Equal contribution

## II. Related works

### A. Solvers for OMT

Optimization modulo theories has been explored over the last decade, but there are not many publicly available solvers. Two prominent solvers are OptiMathSAT [4], which applies a binary search approach to discovering the optimal solution, and *vZ* [5], 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 reencodes the problem using sorting networks to decouple the Boolean reasoning from the arithmetic solving. *vZ* 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 in practice.

Similar to OMT, Inez [6], 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.

### B. 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. However, like all tree search algorithms, branch and bound can incur an exponential worst case complexity for adversarial node selections.

In Balcan et al.[7], 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 discrete input variable constraints.

Another approach taken by Wu et al. [8] 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. [9] extends Balcan et al. [7]'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.

### C. Neural guidance for Combinatorial Applications

Using neural networks to solve TSP [10] has been extensively studied dating back to the development of Hopfield neural networks in 1985 [11]. More recent efforts such as Selsam et al. [12] 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 [13], 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.

## III. Preliminaries

In this section, we provide some background on Optimization Modulo Theories and graph neural architectures used in Ashera.

### A. Optimization Modulo Theories (OMT)

Optimization Modulo Theories (OMT) extends SMT, guaranteeing 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 $\delta$ must be specified.

A Satisfiability Modulo Theory (SMT) problem decides the satisfiability of a first-order formula within a theory [14]. We focus in this exposition on the theory of Linear Integer Arithmetic (LIA), but note that SMT and OMT extend to other theories including 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

$$clause_j \triangleq \bigvee_i atom_i \qquad formula \triangleq \bigwedge_j clause_j$$

. 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 $\alpha$ 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/problem, the satisfiability task only requires reasoning about feasibility. In contrast, an OMT solver must continue to 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 in OMT.

### B. 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 $\alpha$. 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 $\alpha$ deselects the clause. 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 $\alpha$ results in the selection of which clause must hold.

It's worth noting that to solve ILP, the canonical approach relies on making repeated calls to an LP solver which relaxes the integer constraint. This gives a lower bound to the optimization objective as the integer solution is also a LP solution. Thus, it must be strictly less optimal. The most famous algorithm for ILP is branch-and-bound, in which the solver repeatedly picks a variable to branch on, call this variable $v$. This branching consists of resolving the LP with the added constraint that $v$ is either rounded up or rounded down. Eventually once all branches of the tree are either explored or pruned due to having strictly worse lower bound than the best seen solution, the solver will terminate, returning the optimal solution.

Notably, ILP solvers struggle to perform when there are many optimal or near optimal solutions. The existence of multiple disjoint global optima requires hinders the branch-and-bound algorithm from terminating without exploring all the symmetric branches. In practice, an expert is called upon to tune the encoding by introducing symmetry breaking clauses based on an extensive literature and past experience working with symmetry in solvers. Symmetry breaking clauses simplify the problem by placing a preferences between otherwise indistinguishably optimal solutions. This can reduce the solve time from days to a matter of minutes.

### C. 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 [15] 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. [16], these models can be thought of as approximating a learned message passing algorithm over the graph structure, where messages are restricted to the connectivity provided by the edges in the graph. 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 learned function parameterized by $\theta$ 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 (GCN), developed by Kipf and Welling [17], 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 [18] to predicting molecule properties [16] and within combinatorial optimization has been used for network planning [13] and chip placement [2].

**Graph encoding for constrained programming.** As done in Gasse et al. [19] and Nair et al. [9], 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 the 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.

## IV. Method Overview

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

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 initial feasible assignment is selected by the neural diver and how it is trained in Section VI.

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 alternates 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 requirements of the optimizer and verifier in Section V. When the engine's 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.

## V. 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 VI, we introduce the full Ashera algorithm, including the neural diver. We highlight our insight on using Logical Neighborhood Search before justifying the soundness of the clause dropping strategy to improve performance.

### A. Logical Neighborhood Search

Efforts like Wu et al. [8] 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 $\epsilon$-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.

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. 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, $\delta$, can be set depending on the user's domain expertise. For our integer example, $\delta = 1$ is natural.

### B. Clause 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.** *Let $L^+$ and $L^-$ be the sets of true and false (theory) literals that appear in a satisfiable SMT formula $\phi$ in NNF over LIA. Literals $l \in L^+$ define a valid solution space for variables in $\phi$. That is, $l \in L^-$ can be dropped while maintaining soundness.*

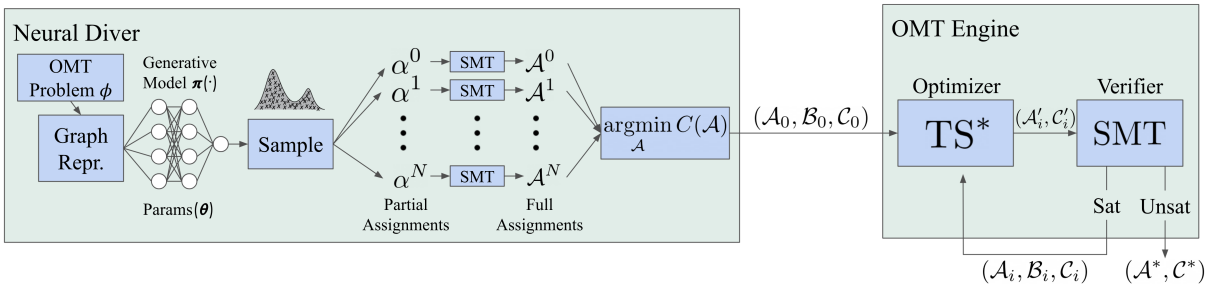


Fig. 1. Ashera workflow. The neural diver serves as a warm-starter providing an initial low cost feasible solution. After neural diving, the OMT engine in Ashera alternates between a *Optimization-aware Theory Solver* (TS*) for optimization and an SMT solver for verification. Each invocation of TS* returns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

**Algorithm 1 Ashera: OMT Solver**

**Input:** $\phi$: 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** $\alpha$ **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** $\mathcal{A}^*, \mathcal{C}^*$

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

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.

## VI. Neural Diving

Inspired by work by Nair et al. [9], we adopt a neural diver which can be thought of as a warm-starter that identifies promising initial feasible solutions.

### A. Graph Represenation

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 [19]. 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 (ignoring uniquifying indexes) 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 is then connected as leaves in an adjoining tree representing the clauses and final formula.

Taking from the approach in [9], 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, which encodes the coefficients as edge weights.

In contrast to the encoding for Nair et al. [9], OMT also includes disjunctions over the literals. As such, we encode the disjunction as a tree over the literal nodes. Adding disjunction and conjunction nodes allows information to pass between clauses in rounds of message passing.

### B. 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) [17]. 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:

$$\mathcal{L} = -\sum_{i=1}^{m} x_i^* \log p(x_i|G) \quad (1)$$

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.

### C. Partial Assignment Warm-Start

When the neural diver is used to solver a problem of interest, the diver makes a prediction based on the input problem $G$. This inferences results in an estimated probability $P(x|G)$ returned as output logits. We sample variable assignments based on these logis and use the entropy of the logits to estimate confidence. If the entropy is larger than $H$, we abstain from making the assignment. This results in partial assignments $\alpha^i$ 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 [20] 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 and in the case that it does not find any feasible solution, the OMT engine runs SMT first to get a feasible assignment. By default Ashera uses $T = 5s$, $K = 5$, and $H = 1$, but these parameters can be tuned on a validation set in practice for each problem family.

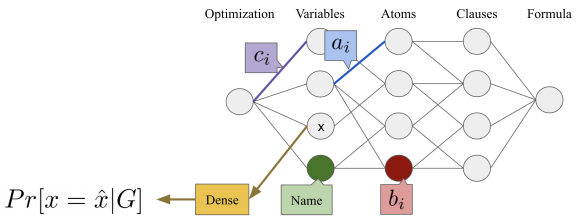


Fig. 2. Graph representation. We represent the OMT problem as a graph with edges connecting variable nodes and constraint/cost nodes. The logic structure is represented similarly to an abstract syntax tree (AST), with the AST leaf nodes coinciding with the atomic constraint nodes.

## VII. Experiments

### A. Setup

For the evaluation experiments, we used c5.12xlarge AWS cloud instances, which have 3.6 GHz 2nd generation Intel Xeon processors with 48 vCPUs and 96 GiB of RAM. For training, we used two Xeon Gold 6226 CPUs, 128 GB of RAM, and two 24GB Titan-RTX GPUs. The models were evaluated without GPU assistance in order to provide a fair comparison, but we expect improved performance if accelerators are available at inference time. We implemented parallel assignment search using Ray 1.10 [20] a distributed programming framework. Our evaluation uses the following solver versions as baselines and as subroutines in Ashera: Gurobi v9.5.1 [3], Z3 4.8.16 [21], and OptiMathSAT 1.7.3 [4].

### B. Datasets

Unlike existing OMT benchmarks which provide several dozen unrelated hard OMT problems, for each family of problems in our benchmark, we providing instances labeled with their optimal assignment. This allows for evaluating learned-OMT solvers that can train on the problem family and evaluate on a holdout set of unseen problems from the same family.

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 $\alpha$ to determine which clause in the disjunction is enforced as a ILP constraint. This in effect increases the dimension of the problem and requires the ILP solver to effectively optimize over every disjunctive branch simultaneously.

For this work, we look at two families of OMT problems: 1) DAG job scheduling, and 2) multi-agent traveling salesman problem (TSP). We formalize these problem families with their encodings in the following sections.

### C. 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 non-trivial to find any feasible schedule, much less an optimal one.

This family of problems appears in both the workflow management platform Apache Airflow [22] and in the DAG scheduler used for the dynamic deadline-driven execution model for self driving [23].

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 II. 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.

### D. 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 B:

- **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 $\tau$ 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.

**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.

## VIII. Results

In this section, we present our empirical analysis of existing OMT tools, *vZ*, OptiMathSAT, Gurobi (ILP with Big M), and Ashera, our proposed solver. 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 scheduling and multi-agent TSP.

### A. Task Scheduling for Directed Acyclic Graphs

In Table I, we provide summary information on the task scheduling benchmark based on the encoding in Section VII-C. Naturally problems with more tasks result in both more constraints and more variables. We generate closely related DAG scheduling problems using the same problem encoding. Further, we in total generate 23,136 instances varying in the number of tasks and the number of 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 I. In this section, we 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 VIII-C. We use $T = 1m$ for 10 Tasks and default values otherwise.

We report the performance of baseline solvers and Ashera in Table II categorized by the number of tasks in the problem instance. Although problems were generated for 11 to 15 task we do not include these in our evaluation as all the solvers timeout on problems of that scale. The table indicates of existing baselines OptiMathSAT has the best performance until 8 tasks and then times out on all instances. Gurobi however,

TABLE I
Scheduling Benchmark Summary.

| Number of Tasks[1] | Number of Training Cases | Number of Test Cases | Average Variable Count | Average Constraint Count | Deadline |
|---|---|---|---|---|---|
| 5 | 811 | 90 | 16 | 48 | 50 |
| 6 | 1459 | 162 | 19 | 63 | 57 |
| 7 | 2383 | 264 | 22 | 80 | 65 |
| 8 | 3630 | 403 | 25 | 99 | 72 |
| 9 | 5250 | 583 | 28 | 120 | 80 |
| 10 | 7291 | 810 | 31 | 143 | 87 |

TABLE II
OMT SOLVER PERFORMANCE ON SCHEDULING.

| Number of Tasks | Average Runtime in Seconds (Number Solved in 20 min ) Gurobi | *vZ* | OptiMathSAT | Ashera |
|---|---|---|---|---|
| 5 | 0.035 (90) | 0.42 (90) | **0.0041** (90) | 2.50 (90) |
| 6 | 0.094 (162) | 0.72 (162) | **0.017** (162) | 2.50 (162) |
| 7 | 0.65 (264) | 3.86 (264) | **0.19** (264) | 2.70 (264) |
| 8 | 7.0 (403) | 140 (403) | **2.53** (403) | 3.58 (403) |
| 9 | 66 (583) | Timeout (0) | Timeout (0) | **17.71** (583) |
| 10 | 823 (810) | Timeout (0) | Timeout (0) | **144.50** (810) |

TABLE III
MULTI-AGENT TSP BENCHMARK SUMMARY.

| # Waypoints per Cluster | Number of Training Cases | Number of Test Cases | Average Variable Count | Average Constraint Count |
|---|---|---|---|---|
| 3 | 2500 | 11 | 137 | 428 |
| 4 | 2500 | 11 | 211 | 654 |
| 5 | 2500 | 11 | 301 | 928 |
| 6 | 2500 | 11 | 401 | 1250 |
| 7 | 2500 | 11 | 519 | 1620 |

TABLE IV
OMT SOLVER PERFORMANCE ON MULTI-AGENT TSP.

| # Waypoints per Cluster | Average Runtime in Seconds (Number Solved in 20 min ) Gurobi | *vZ* | OptiMathSAT | Ashera | Ashera Cold-start |
|---|---|---|---|---|---|
| 3 | 4.75 (11) | **0.13** (11) | 0.20 (11) | 2.83(11) | 3.78 (11) |
| 4 | 98.28 (11) | **0.62** (11) | 1.05 (11) | 6.44(11) | 7.57 (11) |
| 5 | Timeout (0) | **4.53** (11) | 7.84 (11) | 26.2(11) | 26.52 (11) |
| 6 | Timeout (0) | Timeout (0) | **75.88** (11) | 92.9(11) | 95.2 (11) |
| 7 | Timeout (0) | Timeout (0) | 865.46 (8) | **715.48(10)** | 737.73 (10) |

TABLE V
ABLATION ON TASK SCHEDULING PROBLEMS.

| Number of Tasks | Average Runtime in Seconds (Number Solved in 20 min ) Ashera | Cold-start | Holdout | Curriculum |
|---|---|---|---|---|
| 5 | 2.50 (90) | **0.39** (90) | 2.73 (90) | -[2] |
| 6 | 2.50 (162) | **0.40** (162) | 2.74 (162) | 2.76 (162) |
| 7 | 2.70 (264) | **0.48** (264) | 2.82 (264) | 2.83 (264) |
| 8 | 3.58 (403) | **1.7** (403) | 5.34 (403) | 3.83 (403) |
| 9 | 17.71 (583) | 25 (583) | 18.83 (583) | **14.85** (583) |
| 10 | 144.50 (810) | 565 (810) | **144.37** (810) | **144.37** (810) |

continues to solve instances solving instances with 9 and 10 tasks where *vZ* and OptiMathSAT timeout. Ashera in contrast is 3.7x and 5.7x faster than Gurobi on problems with 9 and 10 tasks, respectively.

### B. Multi-agent TSP

In our benchmark (summarized in Table III), we generate 2500 instances of TSP with two clusters of waypoints arranged in a polygon, and additionally 11 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, we always select the first waypoint to be the starting point for the vehicles. Table IV shows Ashera outperforms all three baselines on the largest problems, solving 2 more problems on 7 waypoints per cluster. In Table IV, we report Ashera's performance when trained solely on instances with 3 waypoints per cluster regardless of test set. We use $H = 0.68$ and otherwise use default parameters.

### C. Ablations

We explore how Ashera behaves with three key ablations on the scheduling problem to demonstrate the benefits of neural guidance for the OMT setting: 1) **Holdout.** We look at the performance of Ashera when tested on problem sizes that do not appear in training. 2) **Curriculum.** We look at the performance of Ashera when trained on smaller problems and asked to generalize on larger problems. 3) **Cold-start.** We look at Ashera's cold-start performance without the neural diver as described in Section V.

**Generalization on problem size.** In the main evaluation, we train and test Ashera on problems of the same number of tasks. In the holdout ablation, we train the neural diver on all other problems of 5 to 10 tasks except for the number of tasks on which it will be evaluated. We see that it performs slightly worse on problems of 5 to 9 task and comparable on task of size 10. Even though the model sees more examples of scheduling problems, the ablation indicates that on smaller problems there are instances where the diver made better predictions when it was trained on the same number of tasks than when it is asked to generalize.

**Generalization to Larger Problems.** We further evaluate the ability of Ashera to train on smaller problems and generalize to larger problems. Unlike the holdout ablation, Ashera is trained only on problems that are smaller than the test-time number of task. We see that it performs comparably to holdout and is sometimes outperforming holdout likely due to less overfitting. This ablation indicates that Ashera can be trained on smaller problems which are easier to find optimal solutions to scale to larger problems of interest.

**Cold-Start Ashera.** We ablate the learning component of Ashera and see that Ashera performs faster on scheduling problems with 5 to 8 task. We attribute this to a 2 second overhead incurred in order to perform neural network inference on CPU and initialize Ray [20]. We leave for future work improvements afforded by hardware accelerators such as GPUs.

For multi-agent TSP, Table IV shows neural diving provides modest improvement compared to cold-start due to larger number of variables and larger number of constraints.

## IX. CONCLUSIONS

Our work presents Ashera, a neural OMT solver, which performs up to 5.7x faster than Gurobi on problems which cause traditional OMT solvers to reach a timeout at 20 mins. As OMT problems solved in 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.

[1]We generated scheduling benchmarks of 11 to 20 tasks but all methods timeout with 20 minutes at 11 tasks.

[1]We generated multi-agent TSP benchmarks of 8 to 10 waypoints per cluster but all methods timeout with 20 minutes at 8 waypoints per cluster.

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

## REFERENCES


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

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

[3] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: https://www.gurobi.com

[4] R. Sebastiani and P. Trentin, "Optimathsat: A tool for optimization modulo theories," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, pp. 447–454.

[5] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "$\nu$z - an optimizing smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 194–199.

[6] P. Manolios, J. Pais, and V. Papavasileiou, "The inez mathematical programming modulo theories framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, pp. 53–69.

[7] M. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," *CoRR*, vol. abs/1803.10150, 2018.

[8] Y. Wu, W. Song, Z. Cao, and J. Zhang, "Learning large neighborhood search policy for integer programming," in *Advances in Neural Information Processing Systems*, 2021.

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

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

[11] J. J. Hopfield and D. W. Tank, ""neural" computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–152, 2004.

[12] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "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*, 2019.

[13] H. Zhu, V. Gupta, S. S. Ahuja, Y. Tian, Y. Zhang, and X. Jin, "Network planning with deep reinforcement learning," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. Association for Computing Machinery, 2021, p. 258–271.

[14] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, ch. 26, pp. 825–885.

[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

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

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

[18] W. Fan, Y. Ma, Q. Li, J. Wang, G. Cai, J. Tang, and D. Yin, "A graph neural network framework for social recommendations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 5, pp. 2033–2047, 2022.

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

[20] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[21] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[22] [Online]. Available: http://airflow.apache.org/

[23] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: A dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*. Association for Computing Machinery, 2022, p. 453–471.


## APPENDIX

### A. 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** $(\alpha_{i,j} = 1, \beta_{i,j} = 0)$ **:**
$$s_j + e_j - s_i \leq M(1 - \alpha_{i,j}) + M(1 - \beta_{i,j})$$

### B. Multi-Agent Traveling Salesman Problem 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
- $\gamma_v$ - 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,

$$\forall v \in V. \; o_{h,v} = 0$$

and also

$$\forall w \in W \setminus \{h\}. \bigvee_{w' \in W, v \in V} ((\mathbf{M}_{v,w',w} \vee \mathbf{H}_{v,w',w}) \wedge o_{w',v} = o_{w,v} - 1)$$

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:

$$\forall v \in V. (u_v \leftrightarrow \bigvee_{w,w' \in W} \mathbf{M}_{v,w,w'} \vee \mathbf{H}_{v,w,w'})$$

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:

$$\sum_{v \in V, w \in W, w' \in W \setminus \{h\}} \mathbf{M}_{v,w,w'} \cdot \tau_{v,w,w'} = t$$