Scalable Verification of GNN-Based Job Schedulers

HAOZE WU, Stanford University, USA
CLARK BARRETT, Stanford University, USA
MAHMOOD SHARIF, Tel Aviv University, Israel
NINA NARODYTSKA, VMware Research, USA
GAGANDEEP SINGH, University of Illinois at Urbana-Champaign, USA

Recently, Graph Neural Networks (GNNs) have been applied for scheduling jobs over clusters, achieving better performance than hand-crafted heuristics. Despite their impressive performance, concerns remain over whether these GNN-based job schedulers meet users' expectations about other important properties, such as strategy-proofness, sharing incentive, and stability. In this work, we consider formal verification of GNN-based job schedulers. We address several domain-specific challenges such as networks that are deeper and specifications that are richer than those encountered when verifying image and NLP classifiers. We develop `vegas`, the first general framework for verifying both single-step and multi-step properties of these schedulers based on carefully designed algorithms that combine abstractions, refinements, solvers, and proof transfer. Our experimental results show that `vegas` achieves significant speed-up when verifying important properties of a state-of-the-art GNN-based scheduler compared to previous methods.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: Formal Verification, Neural Networks, Graph Neural Networks, Cluster Scheduling, Abstract Interpretation, Forward-backward Analysis

ACM Reference Format:
Haoze Wu, Clark Barrett, Mahmood Sharif, Nina Narodytska, and Gagandeep Singh. 2022. Scalable Verification of GNN-Based Job Schedulers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 162 (October 2022), 30 pages. https://doi.org/10.1145/3563325

1 INTRODUCTION

Designing efficient job scheduling for multi-user distributed-computing clusters is a challenging and important task [Barroso et al. 2013]. One of the main evaluation metrics of a schedule is performance, for example optimizing job completion time on a job profile. However, the user expectation typically requires that the scheduler satisfy a number of important properties beyond performance, such as strategy-proofness, sharing incentive, and stability [Ghodsi et al. 2011; Zaharia et al. 2010]. If a scheduler lacks any of these properties, the result could be catastrophic, potentially costing millions of dollars at scale. For example, if the scheduler is not strategy-proof (meaning that users can benefit from misrepresenting their job attributes), the users would be incentivized to manipulate their jobs to get them scheduled earlier than they are supposed to. The result could be long waiting times for all users or inefficient overall operation of the cluster [Zaharia et al. 2010].

Authors' addresses: Haoze Wu, Department of Computer Science, Stanford University, USA; Clark Barrett, Department of Computer Science, Stanford University, USA; Mahmood Sharif, School of Computer Science, Tel Aviv University, Israel; Nina Narodytska, VMware Research, USA; Gagandeep Singh, School of Computer Science, University of Illinois at Urbana-Champaign, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/10-ART162
https://doi.org/10.1145/3563325

Recently, a class of job schedulers [Mao et al. 2019; Park et al. 2021; Sun et al. 2021] based on Graph Neural Networks (GNNs) were shown to achieve significant performance improvement over schedulers using hand-crafted heuristics. However, whether these GNN-based job schedulers possess essential properties is not known and, more importantly, there are no tools available to check whether these properties hold. Formally guaranteeing these properties is known to be difficult and until now has only been achieved for simple hand-crafted policies [Shenker and Stoica 2013]. Introducing techniques for proving or disproving such properties for GNN-based schedulers would allow system designers to better evaluate the policies implemented by these schedulers, making adjustments so that the scheduler satisfies users' expectations without sacrificing the performance too much. However, GNNs' decision-making processes are complex and opaque, making it challenging to formally validate these properties.

In this work, we focus on the formal verification of GNN-based job schedulers, which, to the best of our knowledge, has not been considered in prior work. In particular, given a specification over a GNN-based job scheduler, our goal is to either formally prove the specification holds or disprove it with a counter-example. While there is a growing body of work on formally analyzing and verifying properties of deep neural networks applied in the vision, robotics, and natural language processing domains, work on formal analysis of ML models in the systems domain has been limited. This may be explained in part by the unique challenges posed by the systems domain, some of which we outline below.

Computation graph with 100+ layers. GNN-based systems, including schedulers, perform a message-passing algorithm as part of the inference stage. While message passing can be unrolled into a sequence of affine and non-linear activation layers, the resulting network is quite deep, typically containing over 100 layers. Existing state-of-the-art verifiers are designed to handle shallower networks (typically < 20 layers) and begin to lose substantial precision [Boopathy et al. 2019; Dutta et al. 2018; Ehlers 2017; Gehr et al. 2018; Huang et al. 2017; Lyu et al. 2020; Müller et al. 2021b; Raghunathan et al. 2018; Salman et al. 2019; Singh et al. 2019a, 2018a, 2019b,c; Tjandraatmadja et al. 2020; Tjeng et al. 2019; Tran et al. 2020; Wang et al. 2018a,b, 2021b; Weng et al. 2018; Wong and Kolter 2018; Wu et al. 2020a; Xiang et al. 2018; Zelazny et al. 2022; Zhang et al. 2018] or scalability [Anderson et al. 2019; Bak et al. 2020; Botoeva et al. 2020; Bunel et al. 2020, 2018; Ehlers 2017; Fischetti and Jo 2017; Fromherz et al. 2020; Henriksen and Lomuscio 2021; Katz et al. 2017, 2019; Khedr et al. 2020; Lu and Kumar 2019; Tjeng et al. 2019; Tran et al. 2020; Vincent and Schwager 2020; Wu et al. 2022b; Xu et al. 2020] with increasing network depth. To deal with this challenge, we propose a new, general framework for iterative forward-backward abstraction refinement that balances the analysis precision and speed for deeper networks.

Rich Specifications. Many desirable properties require reasoning about sets of nodes rather than a single node. For example, one might specify "no task from job A is scheduled before at least one task from job B is finished" for strategy-proofness. Properties like this contain a large disjunction: we need to check the requirement for each task in job A. We propose an abstraction technique that can reason about multiple disjuncts simultaneously to speed up verification of such properties. As with robustness properties, many desirable properties for schedulers can be defined both globally and locally. We focus on the latter which is popular in the neural network verification literature and stronger than empirical evaluation on finite sets of inputs as done in the past for more complex scheduling policies [Kandasamy et al. 2020]. Our framework can theoretically also handle global properties. We discuss the practical difficulties of it in Sec. 7 and leave it as future work.

Sequential decision making. Schedulers perform sequential decisions to schedule tasks. Therefore, to thoroughly analyze the learned schedulers, it is not sufficient to only reason about single-step

input-output properties considered by state-of-the-art verifiers, as a malicious user can craft a job that affects the scheduler's behavior downstream. Therefore, we consider multi-step verification to reason about bounded traces produced by a sequence of scheduling actions from the scheduler. This adds an extra layer of complexity on top of the already challenging single-step verification, as we need to reason about all the different states along different traces, which requires unrolling the system. To address this, we introduce a proof-transfer encoding of the system which only registers incremental changes in the network encoding along the traces. This significantly speeds up complete verification in the multi-step setting.

This work. We present the first approach for formally analyzing state-based and trace-based properties of GNN-based job schedulers. We build general algorithms for single-step and multi-step verification. Our main contributions are:

- We present a new, generic iterative refinement framework for forward-backward analysis of neural networks. Our framework can be instantiated with popular numerical domains such as Zonotope or DeepPoly to iteratively refine the analysis results.
- We provide a novel, tunable node abstraction for a set of node embeddings produced by the GNN to speed up the verification of properties with multiple disjuncts.
- We present an algorithm for multi-step verification based on trace enumeration. To improve speed, we leverage proof transfer to reuse encodings for the parts of the GNN structures that do not change across time steps.
- We provide an end-to-end implementation of our approach in a framework called `vegas` (verification of GNN-based schedulers) and evaluate its effectiveness for checking desirable state-based and trace-based reachability properties for the state-of-the-art GNN-based scheduler Decima [Mao et al. 2019]. Our results show that analysis with `vegas` is significantly more precise and scalable than baselines based on existing state-of-the-art verifiers. Using `vegas` we prove that Decima satisfies the strategy-proofness property in many cases but not always. Thus adjustments in the training procedure are potentially needed to make Decima fully strategy-proof.

2 OVERVIEW

In this section, we first describe our verification workflow and then explain our key technical contributions using small intuitive examples. Formal details are in later sections.

2.1 Verification Workflow

Our verification workflow shown in Fig. 1 has three components: (a) the system to verify; (b) a formal language for specifying properties; and (c) the verification engine `vegas`.

GNN-based scheduling system. GNN arises as a natural solution for learning-based job schedulers on clusters because many clusters (e.g., Spark) encode jobs as directed acyclic graphs (DAGs), with each node representing a computational *stage* consisting of one or more tasks that can be run in parallel. Each node is associated with a feature vector containing all the state information for that node, including the average task duration and the number of remaining tasks. There is an edge from stage v_i to stage v_j if the latter takes the outputs of the former as inputs. That is, v_j cannot be scheduled before v_i is completed. We call a node with no children a *frontier node*. The input to the GNN-based scheduler is a set of jobs to schedule. The output of the GNN is a score p_i for each frontier node v_i, representing the estimated reward if v_i is scheduled next. The node with the highest score is selected to be scheduled.

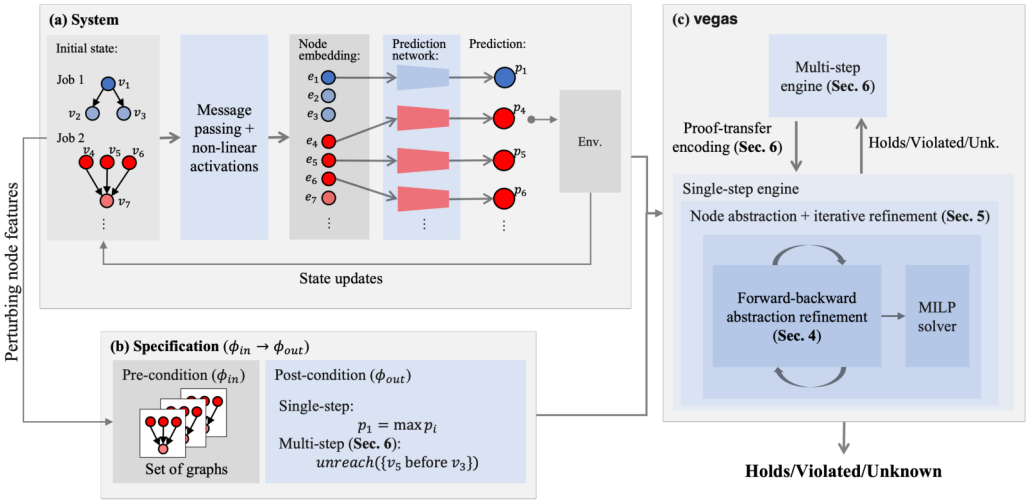

Fig. 1. Overview of our verification workflow. It has three main components: (a) the system to verify; (b) a formal language for specifying properties; and (c) the verification engine vegas.

As is typical in a GNN, the GNN-based job scheduler contains a message passing component which computes a latent representation (i.e., an embedding) e_i for each node v_i. We define message passing precisely in Sec. 3.1. The score p_i for a frontier node v_i is computed from a prediction network which takes e_i as input. The scheduling action (i.e., the node with the highest score) is reported to the environment, which schedules the reported node and produces a new state (with, for example, nodes removed).

Specifications. We consider a wide range of specifications of the form $\phi_{in} \rightarrow \phi_{out}$. We assume ϕ_{in} is a conjunction of linear constraints over the GNN input features, and consider post conditions ϕ_{out} in both single-step and multi-step settings. Single step post conditions are logical constraints over linear inequalities on the outputs of the network. Multi-step post conditions are defined in terms of unreachability of "bad" *traces* (i.e., sequences of scheduling decisions).

Verifier. Our verification engine vegas has two main components, a single-step engine and a multi-step engine. Motivated by the unique challenges in this verification setting, we propose a forward-backward abstraction refinement framework (Sec. 4) which goes beyond the forward-propagation-only abstract interpretation, as well as a node-abstraction scheme (Sec. 5) for handling disjunctions in the verification query. The multi-step engine runs a trace enumeration procedure that repeatedly invokes the single-step engine. We propose an efficient encoding of the unrolled system referred to as the *proof-transfer encoding* (Sec. 6) which significantly reduces the run-time.

2.2 Forward-Backward Abstraction Refinement

As mentioned earlier, one of the distinctive features in GNN verification is the need to reason about very deep computational graphs resulting from the unrolling of the message-passing procedure. Forward abstract interpretation techniques are less effective here as the imprecision can grow exponentially with increasing depth of the computation graph. We propose to refine the forward abstraction by backward refinement guided by the output constraints. This yields a general forward-backward abstraction refinement loop (Sec. 4). While our ideas are driven to tackle challenges in GNN verification, we formalize and implement the proposed technique in a general manner so that it can be applied for neural networks with different architectures and activations.

Running example. We illustrate the forward-backward abstraction refinement on a pre-trained fully-connected feed-forward neural network with Leaky ReLU activation functions ($\sigma(x) = \max(\alpha x, x)$) shown in Fig. 2. Here α is a hyper-parameter of Leaky ReLU. For numerical simplicity, we assume $\alpha = 0.1$ in this example. We use Leaky ReLU as an example since it is used in the state-of-the-art GNN-based job scheduler Decima which we set out to verify. Note that while the running example uses a feed-forward neural network for simplicity, in practice the architecture of a GNN is much more complex (e.g., contains residual connections). We discuss how to handle forward-backward analysis in the GNN setting in Sec. 4.4.

The network here consists of four layers: an input layer, two hidden layers, and an output layer with two neurons each. The outputs of a (non-input) layer are computed by applying an affine transformation to the last layer's outputs followed by the activation function. The activation function is often not applied at the output layer (also in this example). The values on the edges represent the learned weights of the affine transformations. The values above or below the neurons represent the learned biases (translation values) of the affine transformation. For example, the top neuron in the first hidden layer, x_4, can be computed as $\sigma(x_2)$, where x_2, the pre-activation value of the neuron, is equal to $x_0 + x_1$.

Specification. Let us assume a hypothetical job profile with two disconnected nodes. Suppose their feature vectors (1-dimensional in this case) range from $[0, 1]$. Our goal is to prove that the score for the second node (x_{11}) is always greater than the score for the first node (x_{10}), for any possible values of the two features in the range $[0, 1]$.

Forward abstract interpretation. A typical abstract interpretation on neural networks [Singh et al. 2019a, 2018a, 2019b; Zhang et al. 2018] involves propagating the input set (represented in pre-defined abstraction domain such as Zonotope or DeepPoly) forward layer by layer (via pre-defined abstract transformers) to compute an over-approximation of the reachable output set. The specification holds if the over-approximated output set is disjoint from the bad states ($\neg\phi_{out}$). In this work, we use the DeepPoly domain [Singh et al. 2019b] for forward abstract interpretation, though the refinement technique applies to any sub-polyhedral abstract domain [Singh et al. 2018b]. We show the intervals derived by the DeepPoly analysis in the bottom (blue) box in Fig. 2. The steps to deriving these bounds are omitted here. As shown in the figure, the output bounds derived by DeepPoly are not precise enough to prove the property.

Backward abstraction refinement. Normally, at this point, the abstraction-based analysis is inconclusive and we have to resort to search-based methods (e.g. MILP or SMT solvers) which perform case analysis on the Leaky ReLUs and have an exponential runtime in the worst case. Instead, we observe that the forward abstract interpretation typically ignores the post condition when computing the over-approximation at each layer leaving room for refinement guided by the verification conditions and proving the property without invoking search-based methods. In the case where a complete search is unavoidable, a refined over-approximation still helps to prune the search space.

We illustrate a forward-backward abstraction refinement on our example. We associate two abstract elements, a forward one a_i and a backward one a'_i from the same abstract domain (e.g., DeepPoly) with each neuron x_i. The analysis alternates between forward and backward passes, which respectively update the forward abstract elements and the backward abstract elements. We construct new forward transformers that consider both the forward and the backward abstract elements (Sec. 4), yielding more precise forward analysis. The backward elements are initialized to $\top$ in the beginning, and therefore the first forward pass results in the same results as before. This is followed by a backward pass.

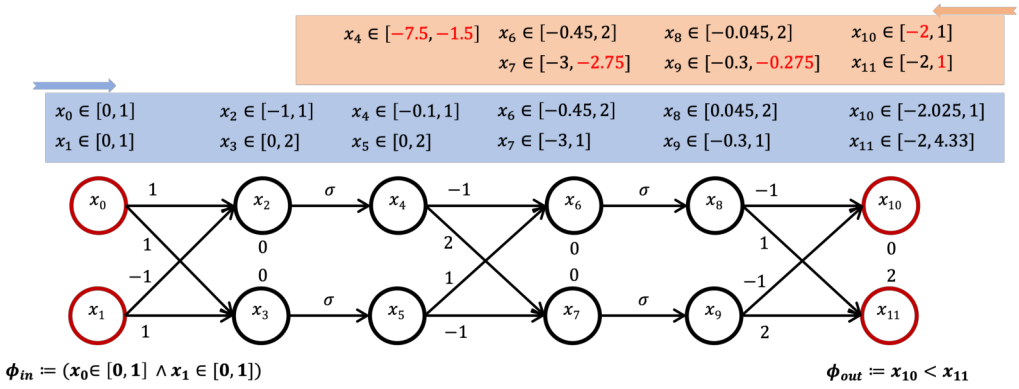

Fig. 2. A toy example for forward-backward abstraction refinement.

The backward analysis starts with the "bad" output set $\neg\phi_{out} \coloneqq x_{10} \geq x_{11}$, which we use to refine a'_{10} and a'_{11}, which are currently set to $\top$. We first compute the bounds of x_{10} and x_{11}, conditioned on the existing bounds and $\neg\phi_{out}$. This yields a tighter lower bound and upper bound for x_{10} and x_{11} respectively. These new bounds are then used to refine the underlying backward abstract element a'_{11}: $a'_{11} = T_{\text{cond}}(a_{11}, x_{11} \in [-2, 1])$, where a_{11} is the forward abstract element updated from the forward pass and T_{cond} is the conditional transformer from the domain. a'_{10} is updated similarly.

We now move to the last hidden layer. Again, we first compute sound intervals for neurons x_8 and x_9 with Linear Programming (LP). For instance, to compute an upper bound for x_9, we can cast the following optimization problem:

$$u_9 = \max_{x_{8:11}} x_9, \text{s.t. } x_{10} = -x_8 - x_9, x_{11} = x_8 + 2x_9 + 2$$
$$x_{10} \in [-2, 1], x_{11} \in [-2, 1], x_8 \in [-0.045, 2]$$

We obtain a tightened bound $x_9 \leq -0.275$. Importantly, now the underlying backward abstract element a'_9 is set to $T_{\text{cond}}(a_9, [-0.3, -0.275])$ where the input-output relationship for the Leaky ReLU is linear (and can be captured exactly by domains like DeepPoly). The exactness significantly improves the analysis precision in the next forward pass.

Note that in the most general form (and in our implementation), two LPs per neuron are called to tighten the bounds. While this incurs overhead, the process is highly parallelizable as neurons from the same layer can be processed independently. More importantly, we observe that this tractable overhead (we prove complexity in Sec. 4) usually pays off in practice on challenging benchmarks which would otherwise require extensive search by a complete procedure.

After refining a'_8 and a'_9, we process x_6 and x_7, where $x_8 = \sigma(x_6)$ and $x_9 = \sigma(x_7)$. Due to the non-linearity of the activation function, a precise encoding of Leaky ReLU results in a Mixed Integer Linear Program, which is in general challenging to solve. Therefore, we encode a sound linear relaxation [Ehlers 2017] of the activation function.

Next, using the same procedure, we derive that $x_4 \in [-7.5, -1.5]$. Note that this is disjoint from the interval derived during forward analysis. Intuitively, this means that for $\phi_{in} \wedge \neg\phi_{out}$ to hold, x_4 must be less than or equal to -1.5, while for ϕ_{in} to hold, x_4 must be between -0.1 to 1. This implies that $\neg\phi_{out}$ cannot hold for any input satisfying ϕ_{in}, and the property is proved without the use of search-based methods.

In the case where the backward analysis does not prove the property by itself, we could perform forward analysis again by taking the refined backward abstract elements into consideration. This could result in a tighter over-approximation of the output set compared to the first forward analysis.

Performing backward analysis again from this tighter over-approximation could in turn result in further refinement. Thus, we could alternate between forward and backward analysis to keep refining the abstractions until either the property is proved or some convergence condition is met. We define this forward-backward analysis formally in Section 4.

2.3 Node Abstraction with Iterative Refinement

If the post-condition ϕ_{out} contains multiple conjuncts, the bad output set $\neg\phi_{out}$ becomes disjunctive. This is a common occurrence in practice as the post-condition often specifies that a set of output neurons all satisfy a certain property P. For instance, the simple post-condition "node v_1 is always scheduled" can be formally specified as $\phi_{out} := \bigwedge \mathrm{p}_1 > \mathrm{p}_i$, where p_i extends over the score of all frontier nodes other than p_1. In this case, the bad output set $\neg\phi_{out}$ becomes $\bigvee \mathrm{p}_1 \leq \mathrm{p}_i$. A naïve approach would analyze each disjunct individually, which becomes expensive, especially when the number of disjuncts is large. However, we observe that the special structure of a GNN used in node prediction tasks allows us to reason about multiple nodes simultaneously.

We illustrate this idea on the weaker post-condition "node v_1 has higher score than frontier nodes in job 2" (in Fig. 1 (a)), that is, $\bigwedge_{i\in\{4,5,6\}} \mathrm{p}_1 > \mathrm{p}_i$. As shown in Fig. 3, a vanilla approach would individually check for the unsatisfiability of the three formulas $\mathrm{p}_1 \leq \mathrm{p}_4$, $\mathrm{p}_1 \leq \mathrm{p}_5$, and $\mathrm{p}_1 \leq \mathrm{p}_6$. However, we observe that in GNNs for node prediction tasks [Wu et al. 2020b], the prediction for a node v_i is often computed by applying the *same* prediction network to the embedding e_i. It is therefore natural to consider an abstraction e' at the embedding level which contains all values that e_4, e_5, and e_6 can take. As illustrated in Fig. 3, suppose $e_4 \in [-2, 2]$, $e_5 \in [-1, 3]$, and $e_6 \in [3, 5]$. An abstract embedding e' can then take values in $[-2, 5]$. If $\mathrm{p}_1 > \mathrm{p}'$ holds, or equivalently, $\mathrm{p}_1 \leq \mathrm{p}'$ is unsatisfiable, then the original post-condition must hold. We state and prove this formally in Sec. 5. On the other hand, if we obtain a spurious counter-example, the analysis is inconclusive and we need to refine the abstraction. One way to refine the abstraction is by reducing the number of node embeddings considered in the abstract embedding. In particular, we heuristically remove the embedding that reduces the volume of the abstraction the most, in this case e_6, and try to reason about $\mathrm{p}_1 > \mathrm{p}_6$ and $\mathrm{p}_1 > \mathrm{p}'$ individually. Now e' is more constrained ($\mathrm{e}' \in [-2, 3]$), and the property is more likely to hold on e'. We can perform this refinement iteratively until either the property is proved or a real counter-example is found. In the worst case, no node abstraction can be performed and we would have to examine each node individually. In practice, it often pays off to speculatively reason about multiple neurons simultaneously, especially when the number of disjuncts is large.

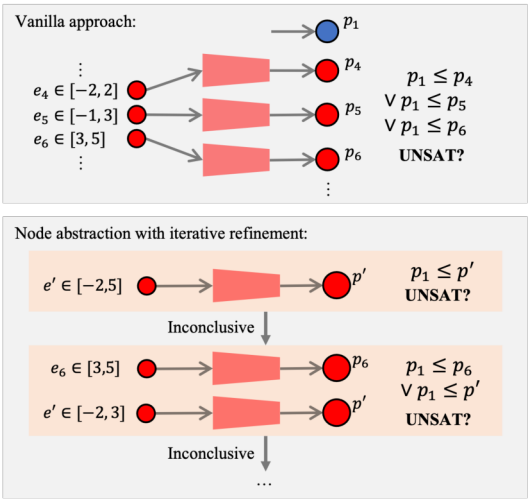

Fig. 3. A toy example for node abstraction.

2.4 Beyond Single-step Verification

Building on top of our single-step engine combining forward-backward refinement and node abstraction, we present a procedure for verifying multi-step properties in the form of trace (un)reachability ($\phi_{in} \longrightarrow \mathtt{unreach}(T)$). As we shall see, this allows us to define meaningful specifications over the job-scheduling system. We illustrate the procedure on the example in Fig. 4. The postcondition we verify states that "v_5 cannot be scheduled before v_3". Our procedure proves this by computing the set

of feasible traces from the initial state by repeatedly invoking the single-step engine. For example, at step 0, we ask the single step engine to check $\phi_{in} \rightarrow (v_1 \geq v_4)$ and $\phi_{in} \rightarrow (v_1 \leq v_4)$. If both can be violated (i.e., v_1 and v_4 can be scheduled), we proceed by computing the reachable traces from v_1 and v_4, respectively, and so forth. During this process, whenever we construct a reachable (partial) trace t, we check whether it matches the prefix of any traces in T. If it does not (e.g., $v_1 \rightarrow v_4 \rightarrow v_3$), we do not need to continue growing that trace as the property must hold for any trace with this prefix. If the partial trace matches exactly with a trace in T (e.g., $v_4 \rightarrow v_1 \rightarrow v_2 \rightarrow v_5$), then the post-condition is violated. Finally, if it is inconclusive, then we need to continue growing the trace.

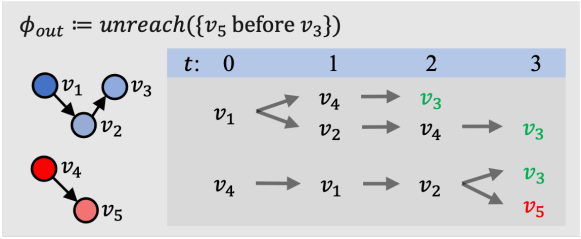

Fig. 4. A toy example for multi-step verification.

When computing the possible next actions from a partial trace, a complete encoding of the system requires unrolling, i.e., an encoding of the system over multiple steps. For example, as shown in Fig. 5, suppose in the first step node v_4 is scheduled; then, in the second step, the graph is updated with one removed node. A naïve encoding would re-encode the entire neural network for step 2 and invoke the single-step engine on this widened neural network to check whether $p_1 @2 \geq p_5 @2$ and $p_5 @2 \geq p_1 @2$ are respectively feasible under the additional constraint that $p_4 @1 \geq p_1 @1$. A naïve encoding that introduces a fresh encoding of the network for each time step quickly becomes intractable. On the other hand, an incomplete encoding that ignores the previous time steps and only encodes the current step can find spurious counter-examples.

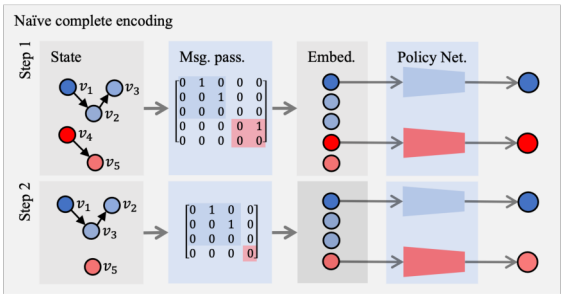

Fig. 5. Example of a naïve encoding of two steps.

In order to obtain an efficient encoding that still tracks all constraints across time steps, we propose building a meta-network that captures the *incremental* changes in the network structure across time steps and reusing the parts that do not change. We refer to this graph as a *proof-transfer network*. Fig. 6 shows the construction of the proof-transfer network for the same two steps as in Fig. 5. In particular, the removal of v_4 only affects nodes in the bottom job (i.e., v_5) and the message passing steps for the other three nodes remain unchanged. This means that at each time step, we can reuse the GNN encoding for all but one job DAG (a disconnected component of the input graph), which results in significantly slower growth in the size of the encoding as the time step increases.

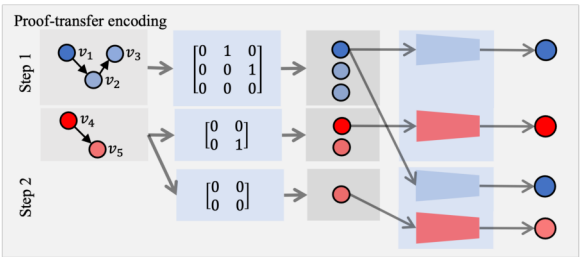

Fig. 6. Example of the proof-transfer encoding of the same two steps as in Fig. 5.

3 PRELIMINARIES

3.1 Graphs and Graph Neural Networks

Definition 3.1 (Graph). We define a graph G with N nodes as a tuple (A, X) where A is an $N \times N$ adjacency matrix and X is the set of node attributes. There is an edge from node v_i to v_j if $A_{ij} = 1$. The neighborhood of a node v_i is defined as $N(\mathrm{v}_i) = \{\mathrm{v}_j \,|A_{ij} = 1\}$. The node attributes $\mathbf{X} \in \mathbb{R}^{N \times d}$ make up a node feature matrix with $x_i \in \mathbb{R}^d$ representing the feature vector of a node v_i. The input to a GNN is a graph $G = (A, X)$. In the case of job scheduling, an input graph can be disjoint, which is useful for modeling collections of jobs.

Graph Convolutional Networks. Graph Neural Networks (GNNs) [Duvenaud et al. 2015; Kipf and Welling 2017; Niepert et al. 2016] are a class of deep neural networks for supervised learning on graphs. They have been successfully employed for node, edge, and graph classification in a variety of real-world applications including recommender systems [Ying et al. 2018], protein prediction [Fout et al. 2017], and malware detection [Wang et al. 2019]. We focus on an important and widely used subset of GNNs called spatial graph convolutional networks (GCN) [Wu et al. 2020b]. Given an input graph, a GCN generates an embedding $\mathrm{e}_i \in \mathbb{R}^d$ for each node v_i by aggregating its own features x_i and all nodes reachable from v_i through a sequence of *message passing* steps. In each message passing step, a node v whose neighbors have aggregated messages from all of their children computes its own embedding as:

$$\mathrm{e}_i = g\Big[\sum_{v_j \in N(v_i)} f(\mathrm{e}_j)\Big] + \mathrm{x}_i$$

where $f(\cdot)$ and $g(\cdot)$ are feed-forward neural networks.[1] In practice, message passing is performed for a fixed number of rounds: in the first round, message-passing steps are performed on a pre-selected initial set of nodes, and in a subsequent round, message-passing steps are performed on neighbors of the nodes that participated in the previous round.

A distinct characteristic of GCNs compared to previous GNN architectures is that there are no cyclic mutual dependencies in message passing, i.e., a GCN can be unrolled. However, unlike the neural networks targeted by existing verifiers, the networks obtained from unrolling are much deeper (100+ layers in practice). Furthermore, the unrolled networks are not simple feed-forward networks but also contain complex residual connections. For simplicity, we assume that the unrolling results in a feed-forward network for the rest of this paper unless specified otherwise. Our approach for handling residual connections is described in Sec. 4.4.

Node regression/classification with GCN. The embeddings e_i after message passing can subsequently be used for different graph analytics tasks. We focus on node prediction tasks where the goal is to predict on nodes indexed by a set Θ. Note that Θ might not include all the nodes in the input graph, as for a scheduler some of the nodes may not be eligible for scheduling at a given time step due to sequential dependencies. The prediction p_i for a node v_i is computed by feeding the node embedding e_i into a feed-forward network h. We allow the flexibility to augment the input to h with an additional non-linear embedding z, computed from the node features and embeddings via a summary network s: $\mathrm{z} = \mathrm{s}(\{(\mathrm{x}_i, \mathrm{e}_i), \mathrm{v}_i \in G\})$. In short, $\mathrm{p}_i := \mathrm{h}(\mathrm{e}_i, \mathrm{s}(\{(\mathrm{x}_i, \mathrm{e}_i), \mathrm{v}_i \in G\}))$.

3.2 Verification of GNNs

We consider verifying a specification ϕ over a GNN F, where ϕ has the form $\phi_{in} \rightarrow \phi_{out}$. The pre-condition ϕ_{in} defines a set of inputs to F, and the specification states that for each input

[1] This definition of message passing covers common forms of GCNs as seen in [Defferrard et al. 2016; Khalil et al. 2017; Kipf and Welling 2017].

point satisfying ϕ_{in}, the post-condition ϕ_{out} must hold. In this work, we limit the form of ϕ_{in} to describing a set of inputs where the graph structure is constant and the node features are defined by a conjunction of linear constraints. Formally, given an input graph $G = (A, X)$ and a GNN $F(\mathrm{m}, \mathrm{h}, \mathrm{s})$ with message-passing component m parameterized by g and f, prediction network h, and summary network s, the concrete behavior of F can be expressed with the following set of constraints (x_i's, e_i's, z, and p_i's are interpreted as real-valued variables):

$$M := \begin{cases} \phi_{in}(\mathrm{x}_1, \ldots, \mathrm{x}_N) \\ \mathrm{e}_1, \ldots, \mathrm{e}_N = \mathrm{m}(\{\mathrm{x}_i, i \in [1, N]\}, A) \\ \mathrm{z} = \mathrm{s}(\{\mathrm{x}_i, \mathrm{e}_i \mid \mathrm{v}_i \in G\}) \\ \bigwedge_{i \in \Theta} \mathrm{p}_i = \mathrm{h}(\mathrm{e}_i, \mathrm{z}) \end{cases} \quad := M_{in} \tag{1}$$

We use M_{in} to denote all outputs before the prediction network h is applied. This will be used when we define the node abstraction scheme in Sec. 5. The verification problem is to check whether $M \rightarrow \phi_{out}$ is valid, or equivalently, whether $M \wedge \neg\phi_{out}$ is unsatisfiable. Under the latter interpretation, the verification problem is to show that under the constraints M, the "bad"states described by $\neg\phi_{out}$ cannot be reached.

Single-step post conditions. We support single-step post conditions of the form $\phi_{simp} := \bigvee_j \psi_j(P)$, where $\psi_j(P)$ is an *atomic* linear constraint over the output variables, i.e., $(\sum_{\ell \in [1,N]} a_\ell \cdot \mathrm{p}_\ell) \bowtie b$, where a_ℓ and b are constants and $\bowtie \in \{=, <, \leq\}$. We refer to this as a *simple* post condition because the "bad" states can be described as a conjunction of linear constraints, and checking ϕ_{simp} amounts to showing that $M \wedge \bigwedge_j \neg\psi_j(P)$. Moreover, we support richer post conditions that state that multiple simple post conditions must hold simultaneously: $\phi_{complex} = \bigwedge_i \phi^i_{simp}$. We describe our novel abstraction to efficiently handle such post conditions in Sec. 5.

3.3 GNN-based Job Scheduling

As a proof of concept, we focus on verifying the state-of-the-art GNN-based job scheduler, Decima [Mao et al. 2019]. Formally, the input to the scheduler is a graph G with K disconnected components $G_1, \ldots, G_K$, representing the current state of the cluster. Each disconnected component G_k is a job DAG. The output of Decima is a single score p_i for each frontier node v_i. We use $\textsc{front}(G)$ to denote the frontier nodes. The node to schedule next is $\arg\max_{\mathrm{v}_i \in \textsc{front}(G)} \mathrm{p}_i$. Our goal is to reason about Decima not only in the single-step setting, but also in the multi-step setting. which we describe next.

Multi-step setting. We consider a setting with an initial state $G = (A, X)$, a GNN-based scheduler F, and a cluster environment $\mathcal{T}(G, \mathrm{v}) \mapsto G'$, which takes the current state G and a node $\mathrm{v} \in \textsc{front}(G)$, and outputs a new state G' representing the new state after v is scheduled. We restrict $\mathcal{T}$ to perform two types of graph updates: 1) removal/addition of nodes; and 2) affine transformations of features of existing nodes: $x'_i \mapsto A\boldsymbol{x_i} + \boldsymbol{b}$. This precisely captures the actual cluster environment for a wide range of initial states. For simplicity, we only consider static job profiles with no new incoming jobs (i.e., only node removal and no node addition). Given G, F, and $\mathcal{T}$, we refer to a finite sequence of scheduling decisions $F(G), F(\mathcal{T}(G, F(G))), \ldots$ as a *trace*.

Multi-step post conditions. We consider multi-step post conditions defined in terms of trace reachability. The post condition ϕ_{out} is of the form $\texttt{unreach}(T)$ where T is a finite set of finite traces of possibly different lengths. The specification states that none of the traces in T are feasible if the initial state satisfies ϕ_{in}. As we will see in Sec. 7, this form of specifications allows us to specify meaningful multi-step properties for the job-scheduler.

4 FORWARD-BACKWARD ANALYSIS

In this section, we describe our forward-backward abstraction refinement framework in more formal terms. We consider an L-layer feed-forward neural network $f\colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$, where n_0 and n_L are the number of input and output neurons respectively. For an input x, we use $f_\ell(x)$ and $f_{\ell:L}(x)$ respectively to denote the network output at an intermediate layer ℓ and all layers between l and L. We consider the affine and the non-linear activation layers as separate. As illustrated in Sec. 2, our key insight is to iteratively refine the abstraction obtained from a forward abstract interpretation with an LP-based backward pass and vice-versa. Alg. 1 shows the pseudocode for our framework. Next we describe each step in greater detail and prove soundness properties.

Algorithm 1 Forward-Backward Analysis.

1: **Input: neural network** f **and specification** $\phi_{in} \to \phi_{out}$
2: **Output: HOLD/VIOLATED/UNKNOWN**
3: **function** forwardBackwardAnalysis(ϕ)
4: $\mathrm{a}^{(1)}, \ldots, \mathrm{a}^{(\mathrm{L})} \mapsto \top, \mathrm{a'}^{(1)}, \ldots, \mathrm{a'}^{(\mathrm{L})} \mapsto \top$
5: **while** $\neg$ stopCondition() **do**
6: $\mathrm{a}^{(1)}, \ldots, \mathrm{a}^{(\mathrm{L})} \mapsto$ forward$(f, \{a^\ell\}, \{a'^\ell\}, \phi_{in})$
7: **if** $T_{\text{cond}}(a^L, \neg\phi_{out}) = \bot$ **then**
8: **return** HOLD
9: $bounds \mapsto []$
10: **for** $\ell = K, K-1, \ldots, 2$ **do**
11: $bounds[\ell] \mapsto$ computeBounds$(f, \ell, \{a^\ell\}, \{a'^\ell\}, \phi_{out})$
12: $\mathrm{a'}^\ell \mapsto T_{\text{cond}}(\mathrm{a}^\ell, (x^{(\ell)} \in bounds[\ell])) \sqcap \mathrm{a'}^\ell$
13: **if** $\mathrm{a'}^\ell = \bot$ **then**
14: **return** HOLD
15: **return** checkSat$(\phi_{in} \wedge (\bigwedge_{\ell \in [1,L]} (\varphi^{(\ell)}_{\text{non-linear}} \wedge \varphi^{(\ell)}_{\text{linear}})) \wedge \neg\phi_{out})$

4.1 Forward Abstract Interpretation

The forward analysis in our framework is generic as it can be instantiated with any sub-polyhedral domain including the popular domains for neural network verification such as Boxes [Wang et al. 2018b], Zonotope [Gehr et al. 2018; Singh et al. 2018a], DeepPoly [Singh et al. 2019b], or Polyhedra [Singh et al. 2017]. We use $\mathbb{A}_n$ to denote an abstract element overapproximating the concrete values of n numerical variables. We require that the abstract domain is equipped with the following components:

- A concretization function $\gamma_n\colon \mathbb{A}_n \to \mathcal{P}(\mathbb{R}^n)$ that computes the set of concrete points from $\mathbb{R}^n$ represented by an abstract element $a \in \mathbb{A}_n$.
- A bottom element $\bot \in \mathbb{A}_n$ such that $\gamma_n(\bot) = \emptyset$.
- A sound abstraction function $\alpha_n\colon \mathcal{P}(\mathbb{R}^n) \to \mathbb{A}_n$ that computes an abstract element overapproximating a region $\phi_{in} \in \mathcal{P}(\mathbb{R}^n)$ provided as input to the neural network. We have $\phi_{in} \subseteq \gamma_n(\alpha_n(\phi_{in}))$ for all $\phi_{in} \in \mathcal{P}(\mathbb{R}^n)$. Note that we do not require α_n to compute the smallest abstraction for ϕ_{in}, however, the input regions considered in our experiments can be exactly abstracted with common domains such as DeepPoly and Zonotope.
- A bounding box function $\iota_n\colon \mathbb{A}_n \to \mathbb{R}^n \times \mathbb{R}^n$, where $\gamma_n(a) \subseteq \prod_i [c_i, d_i]$ for $(\boldsymbol{c}, \boldsymbol{d}) = \iota_n(a)$ for all $a \in \mathbb{A}_n$.
- A sound conditional transformer $T_{\text{cond}}(a, C)$ that for each $a \in \mathbb{A}_n$ and set of linear constraints C defined over n real variables satisfies $\gamma_n(T_{\text{cond}}(a, C)) \subseteq \gamma_n(a)$, i.e., the conditional output does not contain more points than the input a.
- A sound abstract transformer $T^{\#}_{of}\colon \mathbb{A}_m \to \mathbb{A}_n$ for each layerwise operation $o\colon \mathbb{R}_m \to \mathbb{R}_n$ (e.g., affine, non-linear activations, etc.) in the neural network.
- A sound meet transformer $\sqcap$ for each $a, a' \in \mathbb{A}_n$ satisfying $\gamma_n(a \sqcap a') \subseteq \gamma_n(a)$ and $\gamma_n(a \sqcap a') \subseteq \gamma_n(a')$.

Our framework associates an abstract element a^ℓ from the forward pass and an element a'^ℓ from the backward pass with each layer ℓ. Both elements are constructed such that they individually

overapproximate the set of concrete values at layer ℓ with respect to $\phi_{in} \wedge \neg\phi_{out}$ at each iteration of the while loop in Alg. 1. Initially, all elements are $\top$.

Constructing forward transformers. The forward pass shown at Line 6 in Alg. 1 first constructs an abstraction of the input region $a^1 = \alpha_{n_0}(\phi_{in})$. We propagate a^1 through the different layers of the network via a novel construction that creates new *higher order* abstract transformers $T^{\#}_{ofb}$ from existing $T^{\#}_{of}$ for each operation o. For a layer ℓ, the construction of $T^{\#}_{ofb}$ takes $T^{\#}_{of}$ and the abstract elements $a^{\ell-1}, a'^{\ell-1}$ at layer $\ell - 1$ as inputs. Its output is the new forward abstract element at layer ℓ:

$$a^{\ell} = T^{\#}_{ofb}(T^{\#}_{of}, a^{\ell-1}, a'^{\ell-1}) = T^{\#}_{of}(a^{\ell-1}) \sqcap T^{\#}_{of}(a'^{\ell-1}) \tag{2}$$

We next prove the soundness of our construction.

Theorem 4.1. *$T^{\#}_{ofb}$ is a sound abstract transformer, that is, given an input that includes all concrete values at layer $l-1$ with respect to $\phi_{in} \wedge \neg\phi_{out}$, the transformer's output includes all concrete values possible at layer l with respect to $\phi_{in} \wedge \neg\phi_{out}$.*

Proof. Let $S^{\ell-1}$ and S^{ℓ} respectively be the set of concrete values at layer $\ell-1$ and ℓ with respect to $\phi_{in} \wedge \neg\phi_{out}$. For sound abstractions $a^{\ell-1}, a'^{\ell-1}$, we have $S^{\ell-1} \subseteq \gamma_n(a^{\ell-1})$ and $S^{\ell-1} \subseteq \gamma_n(a'^{\ell-1})$. Since $T^{\#}_{of}$ is sound, $S^{\ell} \subseteq \gamma_n(T^{\#}_{of}(a^{\ell-1}))$ and $S^{\ell} \subseteq \gamma_n(T^{\#}_{of}(a'^{\ell-1}))$. Thus $S^{\ell} \subseteq \gamma_n(T^{\#}_{of}(a^{\ell-1})) \cap \gamma_n(T^{\#}_{of}(a'^{\ell-1})) \subseteq \gamma_n(T^{\#}_{of}(a^{\ell-1}) \sqcap T^{\#}_{of}(a'^{\ell-1})) = \gamma_n(T^{\#}_{ofb}(T^{\#}_{of}, a^{\ell-1}, a'^{\ell-1}))$ □

Corollary 4.2. *The output of $T^{\#}_{ofb}$ is included in the output of $T^{\#}_{of}$ for inputs soundly abstracting the concrete values with respect to $\phi_{in} \wedge \neg\phi_{out}$. That is, the output of $T^{\#}_{ofb}$ is at least as precise as $T^{\#}_{of}$.*

(2) invokes the original transformer $T^{\#}_{of}$ twice along with the $\sqcap$ transformer. For most popular domains, $\sqcap$ is asymptotically cheaper than forward, therefore the asymptotic cost of $T^{\#}_{ofb}$ is same as $T^{\#}_{of}$. Since $T^{\#}_{ofb}$ obtained for each layer is sound, the forward propagation produces $\gamma_{n_L}(a^L) \supseteq f(\phi_{in}) \supseteq f(\phi_{in}) \wedge \neg\phi_{out}$ at the output layer. In this section, we assume that the set of bad outputs $\neg\phi_{out}$ can be described as a conjunction of linear constraints. In the next section, we will consider a more general class of $\neg\phi_{out}$ containing disjunctions also. For our restriction here, we can compute $T_{\text{cond}}(a^L, \neg\phi_{out})$, and if it is equal to $\bot$, we have proved that the specification must hold since $\gamma_{n_L}(T_{\text{cond}}(a^L, \neg\phi_{out})) \supseteq f(\phi_{in}) \wedge \neg\phi_{out}$ (Line 7 in Alg. 1).

4.2 Backward Abstract Interpretation

The forward propagation ignores the post condition ϕ_{out} during the construction of the abstract element a^L. A refinement of the abstraction can be obtained by taking ϕ_{out} into consideration. The backward pass shown at Line 10 in Alg. 1 is designed to accomplish this refinement.

The backward pass first updates the backward element at the output layer $a'_L = T_{\text{cond}}(a_L, \neg\phi_{out}) \sqcap a'_L$. It can be seen that this update is sound. The backward pass at a non-output layer k performs two steps. First, we use Linear Programming (LP) to compute refined lower and upper bounds $\boldsymbol{c^{\ell}}, \boldsymbol{d^{\ell}}$, taking into consideration both ϕ_{out} and a linear overapproximation of the network behavior from layer ℓ to the output layer with respect to $\phi_{in} \wedge \neg\phi_{out}$. We then refine the backward abstract element using the domain conditional transformer T_{cond} and the linear constraints $c^{\ell}_i \leq x^{\ell}_i \leq d^{\ell}_i$ for each neuron x^{ℓ}_i in layer $\ell > 1$, i.e., we compute

$$a'^{\ell} = T_{\text{cond}}(a^{\ell}, (\bigwedge_i c_i \leq x^{\ell}_i \leq d_i)) \sqcap a'^{\ell}. \tag{3}$$

We now describe our linear encoding of the network for computing the refined interval bounds $\boldsymbol{c^{\ell}}, \boldsymbol{d^{\ell}}$ of the neurons in layer ℓ. We start with an empty set of constraints and iteratively collect linear

constraints $\varphi^k(\boldsymbol{x^{k-1}}, \boldsymbol{x^k})$ overapproximating the network behavior with respect to $\phi_{in} \wedge \neg\phi_{out}$ for each layer $k > \ell$. If k is an affine layer, then we add constraints of the form $x^k = A^{k-1}\boldsymbol{x}^{k-1} + b^{k-1}$, where $A^{k-1}, b^{k-1} \in \mathbb{R}$ are the learned weights and biases respectively of the affine layer. If k is an activation layer, then we add the constraints from a linear approximation [Singh et al. 2019b] of the layerwise activation $\boldsymbol{x}^k = \sigma(\boldsymbol{x}^{k-1})$. The linear approximation can be obtained directly using the corresponding domain transformer $T^{\#}_{of}$ or a more precise relaxation based on the constraints from the forward abstract element. If k is the output layer, then we add the constraint $\neg\phi_{out}$ to our encoding. For each k, we also add the constraints C^k obtained from the bounding box ι_n of the abstract element a^k at layer k.

Let φ^ℓ denote the conjunction of constraints collected above. To obtain the refined lower and upper bounds for layer ℓ with m neurons, we need to solve the following two LPs for each neuron x^ℓ_i:

$$c^\ell_i = \min_{\substack{\boldsymbol{x^\ell},\ldots,\boldsymbol{x^L} \\ \text{s.t. } \varphi^\ell}} x^\ell_i, d^\ell_i = \max_{\substack{\boldsymbol{x^\ell},\ldots,\boldsymbol{x^L} \\ \text{s.t. } \varphi^\ell}} x^\ell_i \tag{4}$$

While we refine all neurons in the backward step to gain maximum precision, it is possible to tune the cost and the precision of the backward pass by selectively refining only a subset of the neurons. Note that we do not refine the backward element at the input layer, and it is always $\top$ which preserves soundness.

Theorem 4.3. *For each layer ℓ, the refined bounds $\boldsymbol{c^\ell}, \boldsymbol{d^\ell}$ computed by the LP overapproximate the network output with respect to $\phi_{in} \wedge \neg\phi_{out}$.*

Proof. We show that the constraints in the LP overapproximate the network behavior from layer ℓ to the output which guarantees the soundness of the refined bounds. The proof is by induction. For the base case, the LP constraints at the output layer satisfy $\neg\phi_{out} \wedge C^L \supseteq f(\phi_{in}) \wedge \neg\phi_{out}$. For the inductive step, suppose $\bigwedge_{k\in[\ell+u+1,L]}(\varphi^k(\boldsymbol{x^{k-1}}, \boldsymbol{x^k}) \wedge C^k) \supseteq f_{\ell+u+1:L}(\phi_{in}) \wedge \neg\phi_{out}$ holds with $u \geq 0$. For the layer $\ell + u$, the LP will add the conjunction satisfying $\varphi^k(\boldsymbol{x^{\ell+u-1}}, \boldsymbol{x^{(\ell+u)}}) \wedge C^{\ell+u} \supseteq f_{\ell+u:\ell+u+1}(\phi_{in}) \wedge \neg\phi_{out}$ (since both the box constraints from $C^{\ell+u}$ and the constraints from $\varphi^k(\boldsymbol{x^{(\ell+u-1)}}, \boldsymbol{x^{(\ell+u)}})$ overapproximate $f_{\ell+u:\ell+u+1}(\phi_{in})$). Therefore, $\bigwedge_{k\in[\ell+u,L]}(\varphi^k(\boldsymbol{x^{k-1}}, \boldsymbol{x^k}) \wedge C^k) \supseteq f_{\ell+u:L}(\phi_{in}) \wedge \neg\phi_{out}$ and the induction holds. □

Since the bounds computed by the LP are sound, the update (3) is sound at each intermediate layer ℓ. Therefore the backward pass computes a sound approximation at each iteration of the while loop of Alg. 1. Due to the soundness of the backward pass, if we find that a backward element at a layer $\ell > 1$ became $\bot$ after applying (3), then we can soundly return HOLD (Line 13 of Alg. 1).

Theorem 4.4. *For a layer ℓ, let a'^ℓ and a'^ℓ_{new} be the backward abstract elements before and after the refinement respectively. Then, $\gamma_n(a'^\ell_{new}) \subseteq \gamma_n(a'^\ell)$ holds, i.e., the backward pass does not make the backward abstraction less precise.*

Proof. Follows from the definition of T_{cond} and $\sqcap$. □

Corollary 4.5. *Let q be the total number of neurons in the network f, then the complexity of computing (4) for all neurons in layers $1 < k \leq K$ is $O(q \cdot LP(p, q))$ where $LP(p, q)$ is the cost of solving an LP with p constraints defined over q variables.*

4.3 Iterative Forward-Backward Refinement

We perform an iterative refinement procedure in Alg. 1 by repeatedly performing forward and backward analysis till a stopping criteria is met. Each forward-backward pass results in elements

at least as precise as in previous iterations due to Theorem 4.4 and (2). For arbitrary choices of abstract elements and transformers allowed by our framework, the forward-backward analysis is not guaranteed to converge to a fixed point: another iteration of the while loop does not refine the analysis results. In practice, for piecewise-linear activations like Leaky ReLU used in Decima, we terminate the analysis when a refinement round does not fix the phase of any activations. This is usually achieved within 6 iterations.

If, after the forward-backward analysis has finished, no abstract element has been refined to $\bot$, we resort to a non-linear encoding of the network outputs $\bigwedge_{\ell \in [1,L]} \varphi^{\ell}_{\text{non-linear}}$ combined with linear constraints $\bigwedge_{\ell \in [1,L]} \varphi^{\ell}_{\text{linear}}$ from our final abstract elements. For example, we use a MILP encoding for piecewise linear activations such as Leaky ReLU used in Decima which results in complete verification: the property is satisfied if and only if the set of constraints $\phi_{in} \wedge (\bigwedge_{\ell \in [1,L]} (\varphi^{\ell}_{\text{non-linear}} \wedge \varphi^{\ell}_{\text{linear}})) \wedge \neg \phi_{out}$ is unsatisfiable. The non-linear solver benefits in speed from a reduction in the search area/branches due to the precise constraints added from our refined abstraction.

Theorem 4.6. *Alg. 1 is sound, i.e., $\phi_{in} \wedge \neg \phi_{out}$ is unsatisfiable when the algorithm returns* `HOLD`.

Proof. The individual steps in Alg. 1 are sound. □

4.4 Handling GNN Architectures

GNNs often contain residual connections due to message passing. In those cases, we still need to make sure that when performing abstraction refinement for a certain layer k, we have already refined all subsequent layers that take k's outputs as inputs. To obtain this refinement order, we first construct a DAG from the neural network architecture where each node represents a layer and an edge exists from layer k to layer ℓ if the output of the former feeds into the latter. Drawing inspirations from data-flow analysis [Aho et al. 2007], the backward abstraction refinement is conducted in post-order.

5 NODE ABSTRACTION FOR GNN VERIFICATION

Alg. 1 is our core algorithm for verifying a single step property $\phi_{in} \rightarrow \phi_{out}$ where $\neg \phi_{out}$ is of the form $\bigwedge \Sigma a_i \cdot \mathrm{p}_i \bowtie c_i$ (i.e., a conjunction of linear constraints over the output layer) for a feed-forward neural network. However, only allowing this form of bad outputs is restrictive as in practice the bad outputs specified in many properties (e.g., robustness, strategy-proofness) are disjunctive sets. In particular, the output property often specifies that a subset of output neurons Θ all satisfy certain simple post-condition ϕ, i.e.,

$$\phi_{out} := \bigwedge_{\mathrm{p}_i \in \Theta} \phi(\mathrm{p}_i) \tag{5}$$

Existing works often handle this type of post condition by considering each disjunct individually: we can use Alg. 1 (or any procedure that can handle simple post-conditions) to check whether $\phi_{in} \rightarrow \phi(\mathrm{p}_i)$ holds for each output variable p_i. The original property holds if the solver returns `HOLD` every time, and is violated if the solver returns `VIOLATED` for one of the disjuncts. However, this strategy could be inefficient when the number of disjuncts is large. Instead, in this section, we describe a general procedure to efficiently handle post conditions of this form tailored for node prediction/classification tasks in GNNs. We believe our contributions for verification presented here can be adapted to handle GNN application domains other than job-scheduling such as recommender systems and malware detection.

Our key insight is that each score s_i is computed by applying the same transformation to the node embedding e_i, $\mathbf{p_i} := \mathrm{h}(\mathrm{e}_i, \mathrm{s}(\{\mathrm{x}_i, \mathrm{e}_i \mid \mathrm{v}_i \in G\}))$. Therefore, to simultaneously reason about the GNN's output for a group of nodes $\{\mathrm{v}_i \mid i \in \Theta\}$, it is natural to consider an abstraction that

treats $\{e_i, i \in \Theta\}$ as an equivalence class. Given a set of constraints M as defined in (1) that exactly captures the concrete behaviors of the GNN with respect to ϕ_{in}, we can construct an abstraction $\widetilde{M}$ by mapping each constraint "$p_i = h(e_i, z)$" to "$p' = h(e', z) \wedge I(e')$", where e' and p' are free real-valued variables and I is an *invariant* which we will describe next:

$$\widetilde{M} := \begin{cases} M_{in} \text{ (defined in (1))} \\ p' = h(e', z) \wedge I(e') \end{cases}$$

Definition 5.1 (Soundness). We say $\widetilde{M}$ is a *sound* abstraction of M with respect to a simple output property ϕ, if

$$\widetilde{M} \rightarrow \phi(p') \implies M \rightarrow \bigwedge_{i \in \Theta} \phi(p_i).$$

By definition, given a sound abstraction, we can prove the original property by proving a simple specification on $\widetilde{M}$.

Lemma 5.2. *if $M_{in} \rightarrow I(e_i)$ for each $i \in \Theta$, then $\widetilde{M}$ instantiated with I is sound.*

Proof. Suppose $\widetilde{M} \rightarrow \phi(p')$. It follows that $\forall e, M_{in} \wedge I(e) \rightarrow \phi(h(e, z))$. For each $i \in \Theta$, we can instantiate e with e_i and get $M_{in} \wedge I(e_i) \rightarrow \phi(h(e_i, z))$. Since $M_{in} \rightarrow I(e_i)$, we have $M_{in} \rightarrow \phi(h(e_i, z))$. This is equivalent to $M_{in} \wedge (p_i = h(e_i, z)) \rightarrow \phi(p_i)$. Notice that $M_{in} \wedge (p_i = h(e_i, z))$ is a subset of constraints in M. Therefore, $M \rightarrow \phi(p_i)$. □

Node abstractions. While the choice of I is flexible as long as it yields a sound abstraction, there is a trade-off controlled by I between the difficulty of proving $\widetilde{M} \rightarrow \phi(p')$ and the likelihood that this property holds. For example, the weakest invariant is $\top$, which leaves e' unconstrained, making it less likely that $\phi(p')$ holds. On the other hand, the strongest invariant is $\bigvee_{i \in \Theta} e' = e_i$, which just moves the disjunction in the output property to be over e' and does not reduce the computational complexity.

A general recipe for constructing I relies on the forward abstract interpretation. After performing the forward analysis on GNN F with pre-condition ϕ_{in} using the abstract transformers $T_g^{\#}$, for each e_i, we can obtain a (typically) convex region ψ_i from the abstract element a_{e_i} such that ψ_i over-approximates the values e_i can take under the pre-condition ϕ_{in}, or in other words, $M_{in} \rightarrow \psi_i(e_i)$. Next, we can compute a convex relaxation of the union of the convex regions and use that as our invariant I, i.e., $I = \text{conv}(\bigcup_{i \in \Theta} \psi_i)$. In practice, we take ψ_i to be the tightest intervals of e_i, and let I be the join of those intervals.

Algorithm 2 Node abstraction with iterative refinement.

```
1:  Input: a message passing component m, a summary component s and a prediction component h, and a specification φ : φ_in → ⋀_{i∈Θ} φ(p_i)
2:  Output: HOLD/VIOLATED/UNKNOWN
3:  function checkWithNodeAbtraction(m, s, h, φ)
4:      A, C ↦ Θ, ∅
5:      while C ≠ Θ do
6:          a_{e_1}, ..., a_{e_N} ↦ forward(M, φ_in)
7:          I ↦ getConvexRelaxation(a_{e_1}, ..., a_{e_N})
8:          M̃ ↦ createAbstraction(φ_in, m, s, h, I)
9:          r ↦ forwardBackwardAnalysis(M̃, φ(p'))
10:         if r = HOLD then
11:             C ↦ C ∪ A
12:             A ↦ ∅
13:         else
14:             a_{e_k} ↦ pickNodeFromAbstraction(A)
15:             A ↦ A \ {k}
16:             t ↦ forwardBackwardAnalysis(F, φ_in → φ(p_k))
17:             if t = HOLD then
18:                 C ↦ C ∪ {k}
19:             else
20:                 return t
21:     return HOLD
```

Theorem 5.3. *If $M_{in} \rightarrow \psi_i(\mathrm{e}_i)$ for each $i \in \Theta$, then the abstraction $\widetilde{M}$ instantiated by $I = \textsc{conv}(\bigcup_{i\in\Theta} \psi_i)$ is sound.*

Proof. For any e_i where $i \in \Theta$, by the definition of convex approximation $\psi_i(\mathrm{e}_i) \rightarrow I(\mathrm{e}_i)$. Therefore $M_{in} \rightarrow I(\mathrm{e}_i)$ and by Lemma 5.2 the statement holds. □

Note that $\widetilde{M}$ can be viewed as the concrete semantics of a GNN $\widetilde{F}$ with an augmented input e' and one output p'. Under this view, checking $\widetilde{M} \rightarrow \phi(\mathrm{p}')$ is equivalent to checking the specification $\phi_{in}(\mathrm{x}_1, \ldots, \mathrm{x}_N) \wedge I(\mathrm{e}') \rightarrow \phi(\mathrm{p}')$ on $\widetilde{F}$.

Iterative refinement of node abstraction. If $\widetilde{M} \rightarrow \phi(\mathrm{p}')$ is proved, then the original property also holds. Otherwise we cannot conclude that the original property is violated. In that case, we can obtain a refinement of the over-approximation by considering fewer nodes in the abstraction. This yields an iterative refinement procedure as described in Alg. 2. We maintain two sets of indices: A is the set of node indices treated as equivalent in the abstraction and a node index $i \in C$ if $\phi_{in} \rightarrow \phi(\mathrm{p}_i)$ has been proved. Given a specification of the form described in Eq. (5), we start by creating an abstract network F' that treats all nodes in the disjuncts as equivalent. If the simple property $\phi_{in} \rightarrow \phi_{\mathrm{p}'}$ can be proved on F', then we add all node indices in the equivalence class C. If we fail to prove the property, we refine the abstraction by considering fewer nodes in the abstraction. In particular, we heuristically pick one node (Line 13) to remove from A. In practice, we pick the node whose removal results in the largest decrease in the volume of the convex relaxation I. In the next iteration, we obtain a different abstract network F' that abstracts over fewer nodes.

Theorem 5.4. *If $\widetilde{M}$ is a sound abstraction of M and* forwardBackwardAnalysis *is sound and complete, then Alg. 2 is sound and complete.*

Proof. If the specification can be violated, then r at Line 9 must always equal `VIOLATED` and each iteration picks and checks a different conjunct (Lines 14-16). Since forwardBackwardAnalysis is sound and complete, it must return `VIOLATED` for one conjunct. Now suppose the specification holds. Since the relaxation is sound and forwardBackwardAnalysis is sound and complete, C must only contain indices corresponding to conjuncts that hold. Moreover, each iteration must increase the size of C (Lines 11 and 18). Since Θ is finite, the algorithm will return `HOLD` in finite number of steps. □

6 REASONING OVER TRACES

So far we have introduced a verification engine in Alg. 2 for single-step properties with pre-condition ϕ_{in} that can be expressed as a conjunction of linear constraints on the input node features and post condition ϕ_{out} of the form described in (5). Building upon this engine, we now develop an analysis to reason about the system behavior across multiple time steps. In this setting, we want to prove that bad traces from a set T are not feasible starting from any state satisfying ϕ_{in} that can be expressed as a conjunction of linear constraints on the input node features. Next, we develop a baseline algorithm that iterates over traces and performs early punning of safe traces. Then, we discuss the precision and performance trade-off in this procedure and describe an efficient encoding of the system across multiple steps that still preserves completeness.

Multi-step verification with trace enumeration. We first present a multi-step verification procedure in Alg. 3 and then describe our optimizations for improving its speed. At a high-level, the algorithm searches for an initial state in ϕ_{in} that would result in a trace $t \in T$ by trying to compute all possible traces starting from ϕ_{in}. The progress of the search is tracked by a stack (initialized with an emptry trace nil) containing the set of (partial) traces that need to be explored.

During the search, we pick a partial trace t (Line 6) from the stack and check whether it matches any traces in T (Line 7). There is a match between two traces if they have the same length and the same action sequence. There are three outcomes of our check. If there is a match (MATCH), then the search terminates as we have found a (potential) violation of the property, and we return either VIOLATED or UNKNOWN depending on whether the encoding and the base solver (getPossibleActions) is complete. Otherwise, if no trace in T has t as a prefix, then the check returns NOMATCH. In this case, we can conclude any trace with prefix t is not in T and move on to analyze another trace. If this check is inconclusive (i.e., no trace in T is equal to t, but some traces have t as prefix), then we must expand this trace to determine whether there is a potential property violation. That is, we need to compute the possible next actions of the GNN agent conditioned on the initial state ϕ_{in}, the transition system $\mathcal{T}$, and the current trace t (Line 10-18).

Note that an exact encoding of the transition system up to t (Lines 11-15) involves the precise encoding of 1) the network at each time step (Line 12); 2) the action taken at each time step (Line 13); and 3) the updates of the feature vectors (Line 14).An incomplete encoding can be achieved by ignoring the first two components. This amounts to ignoring the previous trace and only encoding the network at the current step. We explore the runtime-precision trade-off of the complete encoding in our experimental section. The algorithm returns HOLD if no traces from T are matched during the enumeration. Notice that T is only used in the MATCH function for checking whether the current trace we are exploring is a "bad" trace. Therefore, in practice, instead of providing T as a concrete set of traces, it is sufficient and relatively easier to provide an implementation of the MATCH function corresponding to the property.

Algorithm 3 Multi-step verification via trace enumeration.

```
1:  Input: a message passing component m, a summary component s, a prediction component h, an initial state G_0, a transition system T, and a specification φ : φ_in → unreach(T)
2:  Output: HOLD/VIOLATED/UNKNOWN
3:  function checkWithTraceEnumeration(m, f, g, G_0, φ)
4:    stack ↦ {nil}
5:    while ¬stack.empty() do
6:      t ↦ stack.pop()
7:      r ↦ match(t, T)
8:      if r = MATCH then return VIOLATED/UNKNOWN
9:      else if r = NOMATCH then continue
10:     else M, G, k ↦ φ_in, G_0, 0
11:       for v_i in t do
12:         M ↦ M ∧ encodeNetwork(m, h, s, G)
13:         M ↦ M ∧ encodeAction(M, v_i)
14:         M ↦ M ∧ encodeFeatureUpdates(T, G, v_i)
15:         G, k ↦ T(G, v_i), k + 1
16:     M ↦ M ∧ encodeNetwork(m, h, s, G)
17:     Θ ↦ candidate(G)
18:     Q ↦ getPossibleActions(M, Θ)
19:     stack ↦ stack ∪ {t :: v_i | v_i ∈ Q}
20:   return HOLD
21: function getPossibleActions(M, Θ, G)
22:   Q ↦ {}, φ(v) ↦ (⋁_{j∈Θ} v_j ≥ v)
23:   for J ∈ G do
24:     φ_out ↦ ⋀_{v_i∈J∩Θ} φ(v ↦ v_i)
25:     Q ↦ Q ∪ checkWithNodeAbstraction′(M, φ_out)
26:   return Q
```

The getPossibleActions method at Line 18 can build on top of any single-step verification engine including, in particular, the procedure introduced in Sec. 5. One instantiation tailored to GNN-based job schedulers is described in Alg. 3. Our goal is to check whether an action v indexed by a set Θ can be scheduled. This can be formulated as checking whether the post-condition that v is not the maximum (Line 22) holds. We use the techniques introduced in Sec. 5 to reason about candidate actions belonging to the same job DAG simultaneously. This is based on the observation that candidate actions corresponding to the same job often have similar verification results. Note that here we use a slightly modified version of Alg. 2 where instead of

returning HOLD/VIOLATED/UNKNOWN we return all disjuncts in the post condition that does not hold. This amounts to modifying Line 20 in Alg. 3 to continue the search instead of returning.

Theorem 6.1. *Alg. 3 terminates if T has finite length traces.*

Proof. Let K be the maximum number of possible actions at a given time step and R be the longest trace length in T. In the worst case, there are a finite number of traces (K^R) to check. This is because the algorithm: (a) does not check the same trace twice (this can be proved by induction on the length of the partial trace t); and (b) does not expand any t whose prefix does not match a trace in T (Line 9 of Algorithm 3). □

Theorem 6.2 (Soundness). *If the encodings (Lines 12,13,14, 16) and* getPossibleActions *are sound, then Alg. 3 is sound.*

Proof. The assumptions guarantee that Q is a super-set of the actual feasible actions. By induction on the length of the trace added to *stack*, we can prove that the traces added to *stack* are a super-set of the actual reachable set of traces. Therefore, if no trace added to the stack matches any trace in T, no actual reachable trace can match any trace in T. □

Proof-transfer encoding. In general, if there are changes in node features or graph structures, the message passing would result in different node embeddings. A naïve complete encoding would re-encode message-passing (and subsequent GNN components) for every step. This quickly becomes too expensive as the number of time steps increases. However, taking a closer look at the message passing scheme, we observe that the effect of the graph structure/feature updates on the message passing is local to the disconnected component where the updates occur. This means that we only need to re-encode disconnected components of the graphs that are updated. In the case of Decima, we observe that between every scheduling event (invokation of the GNN agent), only a small subset of the job DAGs are updated. This results in significant savings in the length of the encoding and runtime as demonstrated by our experimental results.

7 SPECIFICATIONS FOR JOB SCHEDULER

In this section we define the properties we verify for GNN-based schedulers like Decima. We emphasize that our framework allows the user to specify a rich set of verification properties. Here, we focus on two formulations of the strategy-proofness properties to demonstrate the capabilities of our method. We choose to study strategy-proofness as it is not only important in practice but also representative of the general form of specifications that our framework can handle.

Strategy-proofness is a desirable property of schedulers that intuitively means: "a user cannot benefit by mis-representing their need." For example, we expect that the user cannot get their jobs scheduled earlier by requiring more resources for them. If this basic property does not hold, malicious users can mislead the system into stalling all but their jobs. Interestingly, this property holds for simple schedulers such as FIFO (first-in-first-out) and CMMF (Constrained Max-Min Fairness) [Shenker and Stoica 2013]. However, due to the non-interpretable nature of the GNN-based scheduler, strategy-proofness cannot be guaranteed by construction.

Definition 7.1 (Single-step strategy-proofness). Given an initial job profile $G = (A, X)$ containing K jobs $G_1, \ldots, G_K$, suppose the scheduler picks a node from job G_k. For each node feature vector $\mathbf{x_i}$, let x_{id} and x_{it} denote the entries of estimated total duration and the number of tasks, respectively. Let $G_a \in G$ be a job other than G_k (e.g., the job of an adversarial user). Let C and C_a denote the frontier nodes in G and G_a, respectively. The job scheduler is strategy-proof with respect to G and

G_a, where $a \neq k$, if $\forall G' = (A, X')$,

$$\left.\begin{aligned} \bigwedge_{\mathrm{v}_i \in C_a} (\mathrm{x}'_{id} \in [\mathrm{x}_{id}, \alpha_d\, \mathrm{x}_{id}] \\ \wedge\, \mathrm{x}'_{it} \in [\mathrm{x}_{it}, \alpha_t\, \mathrm{x}_{it}] \\ \wedge \frac{\mathrm{x}'_{id}}{\mathrm{x}'_{it}} \geq \frac{\mathrm{x}_{id}}{\mathrm{x}_{it}}) \end{aligned}\right\} := \phi^{sp}_{in}(X')$$

$$\rightarrow \bigwedge_{\mathrm{v}_i \in C_a} \left(\neg\Big(\bigwedge_{\mathrm{v}_j \in C \setminus C_a} \mathrm{p}'_i > \mathrm{p}'_j\Big)\right)$$

where α_d and α_t are scalars (> 1).

Intuitively, the pre-condition specifies that the owner of the adversarial job G_a can increase all the features related to job utilization as well as the average task duration (implied by the third constraint) in the frontier nodes of their job. The α parameters define the level of perturbation that the adversary is allowed. The post condition states that none of the frontier nodes in G_a can be scheduled, thus implying strategy-proofness. Note that the inner constraint in the post condition $\bigwedge_{j \in C \setminus C_a} \mathrm{p}_i > \mathrm{p}_j$ is a simple post condition (by De Morgan's law), thus the post condition has the form described in Eq. 5. Also note that the strategy-proofness property is very different from the adversarial robustness property [Szegedy et al. 2013], which states that the neural network's decision does not change in response to small perturbation in the input feature. In contrast, strategy-proofness allows the network's decision to change as we increase the input features of G_a. The property holds as long as no node from G_a is scheduled next (i.e., the malicious user cannot benefit).

Definition 7.2 (T-step strategy-proofness). Given a concrete initial job profile $G = (A, X)$ containing K jobs $G_1, \ldots, G_K$, let $G_a \in G$ be a job that is not scheduled within T steps starting from G. The job scheduler is T-step strategy-proof with respect to G and G_a, where $a \neq k$, if $\forall G' = (A, X')$,

$$\phi^{sp}_{in}(X') \rightarrow \mathtt{unreach}(\{t \mid |t| \leq T \wedge (\exists\, \mathrm{v} \in C_a, s.t.\, \mathrm{v} \in t)\})$$

The pre-condition is the same as in Def. 7.1. Intuitively, the property states that for a job G_a that is not scheduled in T step in the original trace starting from G, the owner of the job cannot get the job to be scheduled earlier by lying about the amount of work in the job. Note that when T is equal to 1, the definition is equivalent to the single-step strategy-proofness property in Def. 7.1.

Without a verification engine, we could only obtain *empirical* guarantees about both properties by checking a finite number of job profiles within the range specified by the pre-condition. However, our framework allows us to check *all* job profiles in that range, thus obtaining *formal* guarantees.

Local vs. global properties. Similar to previous work in neural network verification, the strategy-proofness properties here are defined locally, i.e., with respect to concrete initial job profiles. It is possible to define and verify a global version of the strategy-proofness property, for a set of initial job profiles with the same graph structure using our framework. However, checking this property requires a second copy of the GNN-encoding, thus doubling the size of the verification problem. Taking a step even further, it is possible to verify that the strategy-proofness holds for all job profiles with less than N nodes, by checking the property with our verification engine for each unique job profile with less than N nodes (there are finitely many of them). Abstraction techniques for reasoning about job profiles with different graph structures as a whole are likely needed to improve the verification time. We leave the verification of global properties as future work.

8 EXPERIMENTAL EVALUATION

We implemented the proposed techniques in a framework called vegas and performed an experimental evaluation by using vegas to check whether the single and multi-step strategy-proofness properties as described in the previous section hold on Decima [Mao et al. 2019]—a state-of-the-art GNN-based job scheduler. There are other GNN-based schedulers available (e.g., [Park et al. 2021; Sun et al. 2021]). We choose Decima as the representative as it is by far the most popular and influential. We note that our techniques are general and applicable to other GNN-based schedulers and properties as discussed above. Feedback from vegas can be used by the system developer to adjust their schedulers to balance different user expectations.

8.1 Implementation

Our implementation of vegas is three-fold, including:

(1) A **GNN-based scheduler module** that takes the graph structure of the job profile and the architecture of the GNN agent, and converts them into an internal representation of a feed-forward neural network (with residual connections) with the node features as inputs and node predictions as outputs. As the front-end of vegas, the module also contains API calls to define pre-conditions and post-conditions of the forms described in Subsecs. 3.2 and 3.3.
(2) A **single-step verification engine** which contains a generic implementation of the algorithms introduced in Secs. 4 and 5. For forward-backward analysis, we use DeepPoly as the abstract domain, and use the linear approximation proposed in [Ehlers 2017] extended to Leaky ReLU for the LP encoding. It is worth noting that unlike the original DeepPoly implementation [Singh et al. 2019b], our forward-backward analysis handles feed-forward neural networks with arbitrary residual connections. This generality is needed even for verifying different properties of the same GNN because the graph structures of the initial state affect the order of message passing and yield neural networks with different architectures. The node-abstraction implementation applies to GNN-based node prediction models generally.
(3) A **multi-step verification engine** that contains an implementation of the trace enumeration procedure described in Sec. 6, which repeatedly invokes the single-step verification engine. We support both complete and incomplete multi-step encodings.

8.2 Experimental Setup

We use the same GNN-architecture and training configurations as introduced in the original work [Mao et al. 2019], and obtain similar performance results as in the original work. The trained network has Leaky ReLU as activation function. Unlike the traditional neural network verification setting, where the neural network size (e.g., number of activations) is fixed, the GNN size depends on the sizes of the input graphs, which we specify later. All experiments are run on a cluster equipped with Intel Xeon E5-2637 v4 CPUs running Ubuntu 16.04. Each benchmark is run with 32 threads and 128GB memory. For single-step verification, each benchmark run is given a 1-hour wall-clock timeout. For multi-step verification, the wall-clock timeout is set to 2 hours.

8.3 Single-step Verification

We first evaluate vegas on single-step verification benchmarks. The main question we pose here is: Does vegas scale to large GNN-based schedulers? To answer this question, we perform an extensive evaluation of all the techniques we proposed. Our results demonstrate a significant performance gain over baselines based on state-of-the-art verifiers.

Benchmarks. We evaluate our proposed techniques on verifying the single-step strategy-proofness property as introduced in Sec. 7. The scalars α's (see Def. 7.1) are set to 20, meaning

the owner of the adversarial job can increase the estimated total duration and number of tasks by at most 20 times for any frontier nodes in the job. We consider job profiles that contain either 5 or 10 jobs, which yields a median of 5845 and 10997 activations (Leaky ReLU) in the encoding respectively. After unrolling, the network has 140 layers (treating affine and activation as separate layers). We heuristically select job profile and adversarial jobs (G_a's) from Decima's test beds that would likely result in challenging verification benchmarks following the steps below:

- Sample initial job states with 5 and 10 jobs from Decima's native test bed[2] using random seeds 0-24.
- For each of the initial job states, we run the simulation environment until 1/3 of the nodes are scheduled. At each step, we rank the jobs by the sum of the scores of the frontier nodes in decreasing order. We record steps where the total scores of frontier nodes in the top job and that in the second top job are close (<0.9).
- We use strategy-proofness properties defined on states corresponding to those steps as the benchmarks for single step verification with G_a being the second top job because these are vulnerable states that make for challenging verification benchmarks.

The resulting initial job profiles are sparse graphs, with each job containing on average 9.2 ± 4.2 nodes and 8.5 ± 4.4 edges.

Configurations. To evaluate our proposed techniques, we consider 4 different configurations: 1) F first tries to solve the problem with forward abstract interpretation and falls back to a complete MILP encoding with DeepPoly bounds via the Gurobi optimizer; 2) F+B1 performs the forward-backward analysis for one iteration (forward, backward, and forward again) and falls back to Gurobi; 3) F+BC is the same as F+B1 except that it performs the forward-backward analysis repeatedly until the stopping condition described in Sec. 4 is met, and 4) A+F+BC runs Alg. 2 on top of F+BC. We note that F is equivalent to ERAN [Singh et al. 2019b] with its optimal configuration for complete verification. We did not compare with off-the-shelf verification tools [De Palma et al. 2021; Katz et al. 2019; Müller et al. 2021a; Singh et al. 2019a; Wang et al. 2021b], because to our knowledge none can handle the complex architecture of GNNs without significant implementation overhead.

We first evaluate the first three configurations on the full benchmark set. The result is shown in Tab. 1. We observe that the two configurations that perform the forward-backward analysis solve significantly more benchmarks than F, which only performs a forward pass, with a gain of 77% and 8% of solved instances for graphs with 5 jobs and 10 jobs, respectively. On the other hand, we observe an incremental gain from performing the forward-backward analysis for 1 iteration to performing it to convergence. While we solve 5 more benchmarks, performing forward-backward analysis until the stopping condition might lead to a non-negligible runtime overhead. For example, the average time to solve a 10-job benchmark increases from 206 seconds to 324 seconds (i.e., an overhead of 2 minutes).

Table 1. Instances solved by different configurations and their runtime (in seconds) on *solved* instances (i.e., runtime for timed-out instances is not included).

# jobs (# bench.)	F		F+B1		F+BC	
	Solved	Time	Solved	Time	Solved	Time
5 (66)	26	32742	46	31147	**49**	38080
10 (232)	207	45662	224	46201	**226**	73295

[2] https://github.com/hongzimao/decima-sim/

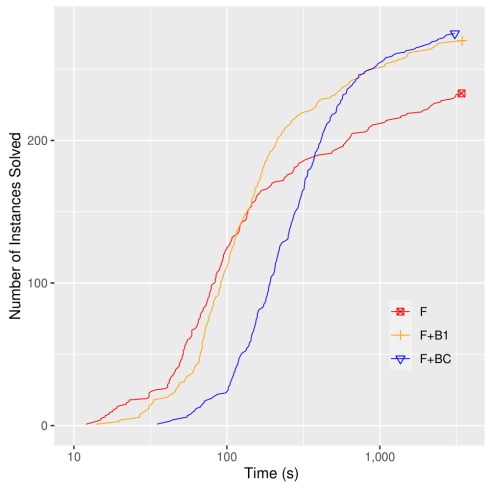

Fig. 7. Cactus plot of the three configurations on the full benchmark set.

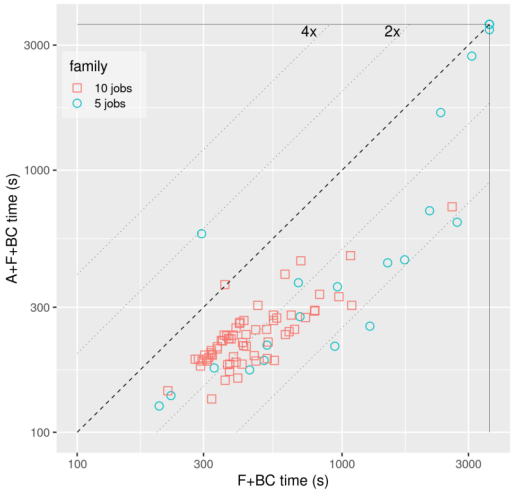

Fig. 8. Runtime of A+F+BC and F+BC on benchmarks with at least 4 disjuncts.

The cactus plot in Fig. 7 sheds more light on the pay-off of the abstraction refinement scheme. While F can solve certain easy instances faster than F+B1 and F+BC, the benefit of the proposed forward-backward analysis becomes evident once the time limit surpasses 100 seconds. On the other hand, F+BC starts to overtake F+B1 when the time limit is above 1000 seconds. To further compare the long-term behaviors of F+B1 and F+BC, we run them on the unsolved benchmarks in Tab. 1 (there are 23 of them) using a longer timeout (2 hours). F+B1 is able to solve 1 of the 23 benchmarks while F+BC is able to solve 4. Based on these results, we recommend to use F+BC when computational resources are not a concern.

Among the 298 verification queries, 257 are proved, 20 are disproved (with counter-examples), and 28 are unknown. This suggests that the current version of Decima is not always strategy-proof, and adjustments in the training algorithm are potentially needed to guarantee it without compromising performance too much.

To evaluate the node abstraction scheme, we focus on the subset of benchmarks where there are at *least 4 frontier nodes* in the adversarial job. This means that the number of disjuncts is at least 4. We run A+F+BC on this subset of benchmarks and compare it with F+BC. As shown in Tab. 2, while only 1 more instance is solved due to the node abstraction, the runtime is reduced significantly. In particular, while both configurations solve the same number of 10-job benchmarks within the time limit, A+F+BC solves them with a runtime reduction of 51.2%. These results clearly demonstrate that using abstract nodes leads to significant computational saving if there are multiple frontier nodes available for scheduling.

Table 2. Instances solved by different configurations and their runtime (in seconds) on solved instances.

# jobs (# bench.)	F+BC		A+F+BC	
	Solved	Time	Solved	Time
5 (26)	18	20600	19	13112
10 (66)	66	33061	66	16028

The scatter plot in Fig. 8 shows the concrete run time of the two configurations on the benchmarks. The node abstraction scheme brings around 2-4x speed-up in a majority of the cases. We notice there are two cases where the verification time is not improved (or even becomes worse). In those cases, the attempts in Alg. 2 to reason about multiple disjuncts simultaneously keep failing, and most disjuncts end up being analyzed individually. This suggests that additional performance gain could be potentially achieved if a more sophisticated heuristic to pick which node to remove from the equivalence class (see Sec. 5) is used.

8.4 Evaluating Forward-Backward Analysis in Isolation

In this section, we further evaluate the effectiveness of the forward-backward analysis as a stand-alone technique. In particular, we pose two research questions:

(1) Is the forward-backward analysis effective as a stand-alone technique (without falling back to a complete solver)? [Yes]
(2) Is the forward-backward analysis useful on other verification benchmarks such as adversarial robustness properties on image classifiers? [Yes]

Benchmarks. We consider the same initial job profiles as described in Sec. 8.3. The specification is also the same except that the scalars α's are set to 8 instead of 20. We choose this value of α because for larger values abstract interpretation alone usually fails to prove the property without invoking the complete solver and for smaller values (<1.5) forward abstract interpretation alone can prove most properties. Additionally, we train two classifiers, $\texttt{MNIST}_1$ and $\texttt{MNIST}_2$, on the MNIST dataset [LeCun and Cortes 2010]. Both are PGD-trained, fully-connected feed-forward, and using Leaky ReLU activations. $\texttt{MNIST}_1$ has 5 hidden layers with 100 neurons per layer. $\texttt{MNIST}_2$ has 8 hidden layers with 100 neurons per layer. We consider standard local l_∞ adversarial robustness properties on the first 100 correctly classified test images. The perturbation bound is set to 0.02 (the inputs are normalized between 0 and 1).

Configurations. We consider three configurations, F', F+B1', and F+BC', which are the same as their counterparts in Sec. 8.3, except that the former do not fall back to complete solvers and instead return UNKNOWN if abstraction interpretation fails to prove the property.

Table 3. Instances solved by different configurations and their runtime (in seconds) on *solved* instances (i.e., runtime for timed-out instances is not included).

Benchmark (#)	F'		F+B1'		F+BC'	
	Solved	Time	Solved	Time	Solved	Time
SP 5 jobs (66)	0	0	22	259	**25**	440
SP 10 jobs (232)	0	0	83	3195	**94**	4815
$\texttt{MNIST}_1$ (100)	31	29	52	72	**56**	135
$\texttt{MNIST}_2$ (100)	24	37	43	100	**46**	175

Experiments. The evaluation results of the three configurations on the aforementioned benchmarks are shown in Table 3. On the verification queries over GNNs, forward abstract interpretation alone (F') is not able to solve any benchmarks, while the two configurations that perform backward abstraction refinement can solve a significant number of the benchmarks. This shows the benefits of forward-backward analysis as a stand-alone technique in improving verification precision and

scalability. It is also worth noting that the effect of performing forward- and backward- analyses multiple times is more evident in this setting, where F+BC' solves 13.3% ($\frac{25+94}{22+83}$) more than F+B1'.

On the adversarial robustness benchmarks, the forward-backward analysis also significantly boosts the verification precision. In particular, F+BC' solves 85% more than F'. This confirms that the forward-backward abstract interpretation can be useful beyond GNN verification.

8.5 Multi-step Verification

We now turn to multi-step verification. Here we ask:

(1) Complete vs incomplete encodings. Is a complete encoding crucial to proving verification queries (due to imprecision of an incomplete encoding)? [Yes]
(2) Does the proof-transfer encoding speed up complete verification? [Yes]

Benchmarks. We evaluate our proposed techniques on verifying the T-step strategy-proofness as introduced in Sec. 7. We choose $T = 5$, which requires enumeration of all possible traces of length 5, and $\alpha = 10$. Initial job profiles are heuristically selected following the steps below with the rationale of identifying challenging benchmarks:

- Sample initial job states with 5 jobs from Decima's native test bed using random seeds 0-99.
- For each initial job state, we run the simulation environment until 1/3 of the nodes are scheduled. At each step, if the score of the top node in the second top job is close to that of the top node in the top job (< 0.9), we use the single-step verification engine to check whether from the pre-condition of the multi-step strategy-proofness properties, the scheduler can choose *multiple* nodes as the next action.
- For each step t satisfying this condition, we add the job state at t, $t-2$, and $t-4$ to the multi-step benchmarks. The resulting benchmarks are guaranteed to have multiple possible traces from the initial set, thus making for more challenging benchmarks.

Experiment. We first evaluate the performance of the three configurations on the task of trace enumeration for 5 steps starting from the pre-conditions specified in strategy-proofness.

Configurations. We consider three configurations of Alg. 3: 1) I does a sound but incomplete encoding of the state as described in Sec. 6; 2) C encodes the states and the state transitions precisely with a naïve unrolling; 3) C+PT also encodes the states and the state transitions precisely but with the proof-transfer encoding described in Sec. 6.

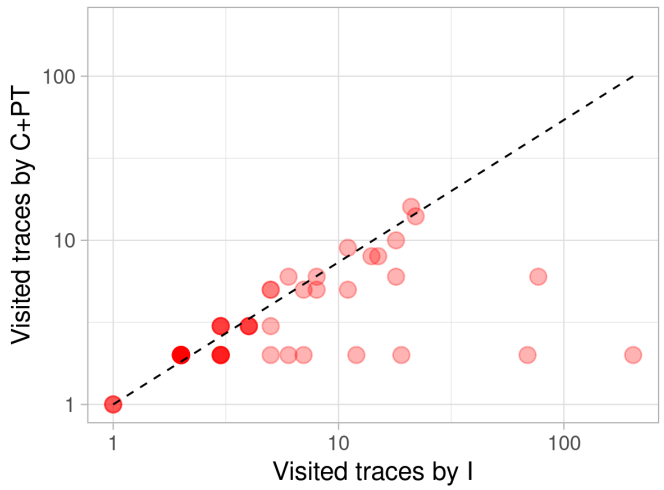

Fig. 9. The number of visited traces by I and C+PT

We found that I and C+PT both terminate on all benchmarks within 2 hours, while C timed out on 15 of the 55 benchmarks. The average runtime of I, C+PT, and C are respectively 350, 1129, and 3582 seconds (treating timed out instances as having runtime 7200 seconds). To understand these results, we recall that all three configurations are based on trace enumeration and pruning. Both C

and C+PT encode the system precisely and therefore do not explore spurious (unrealizable) traces. However, as our experiment shows, the proof-transfer encoding results in significant speedup.

Fig. 9 shows the number of enumerated traces by I and C+PT on each benchmark. While C+PT enumerates the exact set of feasible traces, I enumerates a super-set of feasible traces due to its incomplete encoding. We observe that I includes spurious (i.e., infeasible) traces on 30 out of the 55 benchmarks. The difference in the number of visited traces can get quite large. For instance, for a certain initial condition (bottom right of Fig. 9), there are only two feasible traces but I visits a total of 203 traces. The results emphasize the importance of a precise encoding, as exploring a large number of spurious traces can be computationally prohibitive and might prevent a solver from proving properties, as we see next.

We now turn to verifying the multi-step strategy-proofness properties. To understand whether a precise encoding has benefits over an incomplete encoding, we focus on the 30 initial states where there is a difference in the enumerated traces between the complete and incomplete encodings. The result is shown in Tab. 4. Due to the incomplete nature of I, not only is it unable to generate actual counter-examples, it also proves less properties than a complete procedure like C+PT. In contrast, C+PT solved all queries, and is able to prove 3 more specifications than I. This illustrates the benefit of a complete encoding.

Table 4. I vs. C+PT on the multi-step strategy-proofness benchmarks. We show the number of instances that are proved and disproved respectively, as well as the runtime (s) on *solved* instances.

Prop. (# bench.)	I			C+PT		
	Proved	Disproved	Time	Proved	Disproved	Time
SP (30)	19	0	14704	**22**	**8**	46099

9 RELATED WORK

Most state-of-the-art neural network verifiers perform forward analysis, but a combination with backward analysis is under-explored. [Urban et al. 2020] proposes a procedure specific to fairness properties which uses a forward pre-analysis to partition the input region and a post-condition guided backward analysis to prove the properties for all activation patterns in the input region. [Yang et al. 2021] proposes a fundamentally different abstraction-refinement loop where the backward analysis iteratively refines the pre-condition for DeepPoly analysis. In contrast, we develop a general framework for forward-backward analysis on neural networks that can be instantiated with different abstract domains and prove theoretical results about soundness, monotonicity, and computational complexity of the framework. We also validate our approaches on more complex benchmarks compared with the aforementioned work. Related to abstraction refinement, [Lyu et al. 2020; Ryou et al. 2021] use the post condition to refine the choice of slopes in forward over-approximations but do not consider the post condition as a hard constraint. [Singh et al. 2019c] combines forward abstract interpretation with MILP/LP solving for refinement but only considers the pre-condition, and the refinement is due to the precision gain of the MILP/LP solving.

Verification of RL-driven systems have also gained increasing attention recently [Amir et al. 2021; Sun et al. 2019]. Most recently, [Amir et al. 2021] explores the general use of techniques such as k-induction and invariant inference in those settings while treating the verification procedure as a black box. In contrast, we focus on a challenging and interesting setting of verifying GNN-based job schedulers, where the neural network architecture is more complex and much larger. This motivates the development of a set of new techniques to improve solver's scalability.

The verification of Graph Neural Networks is an important yet under-explored topic. Previous work on GNN-verification focuses on structural perturbations [Bojchevski and Günnemann 2019; Wang et al. 2021a] and robustness properties. In contrast, we focus on a rich set of properties emerging from the scheduling domain. [Wang et al. 2021a] uses random smoothing and therefore gives probabilistic guarantees. On the other hand, [Bojchevski and Günnemann 2019] consider a finite perturbation space (adding/removing finite number of edges) while we focus on infinite perturbation sets defined as linear constraints over the node features.

10 CONCLUSION AND FUTURE WORK

In this work, we proposed the first verification framework for GNN-based job schedulers. This setting poses unique challenges due to deeper network architecture and richer specifications compared to those handled by existing neural network verifiers. We considered both single-step and multi-step verification and designed general methods for both that leverage abstractions, refinements, solvers, and proof transfer to experimentally achieve significantly better precision and speed than baselines. We believe that `vegas` can be used by system developers to check whether different user expectations are met and make adjustments if needed. `vegas` can also be potentially integrated in the training loop of the GNN-based scheduler to guarantee by construction properties specified by the system designer. Similar approaches have been used to create stable neural network controllers in the robotics domain [Dai et al. 2021]. We also believe that the verification benchmarks used in the experiments, which are different from the canonical adversarial robustness queries, would in themselves be a valuable contribution to the research community. Our benchmarks and system are publicly available [Wu et al. 2022a].

We also note that our proposed techniques have different levels of generality. The forward-backward analysis applies to any feed-forward/convolutional/residual neural networks and any abstraction satisfying the conditions in Sec. 4. The node abstraction scheme is generalizable to GNN-based node prediction tasks. The multi-step verification procedure is specific to job-scheduling but could potentially be extended to other RL-driven systems. Exploring the general effectiveness of the proposed techniques would be an interesting direction for future work. There are multiple other promising future directions. First, there is still room to improve the performance of `vegas`: e.g., using more precise abstraction domains or devising a specialized complete procedure that better leverages the problem structure. Second, it would be interesting to evaluate global properties of the GNN-scheduler, which (as discussed in Sec. 7) presents additional research challenges.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback, and Guy Katz for some early discussion on forward-backward analysis. This work was conducted while the first author was an intern at VMWare Research. It was also partially supported by NSF (RINGS #2148583 and NSF-BSF #1814369).

REFERENCES

Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools.* Pearson Education India.

Guy Amir, Michael Schapira, and Guy Katz. 2021. Towards Scalable Verification of Deep Reinforcement Learning. In *2021 Formal Methods in Computer-Aided Design (FMCAD)*. 193–203. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_28

Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. Programming Language Design and Implementation (PLDI)*. 731–744. https://doi.org/10.1145/3314221.3314614

Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T Johnson. 2020. Improved geometric path enumeration for verifying ReLU neural networks. In *International Conference on Computer Aided Verification*. Springer, 66–96. https:

//doi.org/10.1007/978-3-030-53288-8_4

Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.

Aleksandar Bojchevski and Stephan Günnemann. 2019. Certifiable robustness to graph perturbations. *arXiv preprint arXiv:1910.14356* (2019).

Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. 2019. Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3240–3247. https://doi.org/10.1609/aaai.v33i01.33013240

Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. 2020. Efficient verification of relu-based neural networks via dependency analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 3291–3299. https://doi.org/10.1609/aaai.v34i04.5729

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Pushmeet Kohli, P Torr, and P Mudigonda. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21, 2020 (2020).

Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. 2018. A Unified View of Piecewise Linear Neural Network Verification. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/be53d253d6bc3258a8160556dda3e9b2-Paper.pdf

Hongkai Dai, Benoit Landry, Lujie Yang, Marco Pavone, and Russ Tedrake. 2021. Lyapunov-stable neural-network control. *arXiv preprint arXiv:2109.14152* (2021).

Alessandro De Palma, Harkirat Singh Behl, Rudy Bunel, Philip HS Torr, and M Pawan Kumar. 2021. Scaling the convex barrier with active sets. *arXiv preprint arXiv:2101.05844* (2021).

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems* 29 (2016). https://doi.org/10.5555/3157382.3157527

Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*.

David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 2224–2232.

Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 269–286.

Matteo Fischetti and Jason Jo. 2017. Deep Neural Networks as 0-1 Mixed Integer Linear Programs: A Feasibility Study. *CoRR* abs/1712.06174 (2017).

Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein Interface Prediction using Graph Convolutional Networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. 6530–6539.

Aymeric Fromherz, Klas Leino, Matt Fredrikson, Bryan Parno, and Corina Păsăreanu. 2020. Fast geometric projections for local robustness certification. *arXiv preprint arXiv:2002.04742* (2020).

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 3–18. https://doi.org/10.1109/SP.2018.00058

Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. https://doi.org/10.5555/1972457.1972490

Patrick Henriksen and Alessio Lomuscio. 2021. DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Zhi-Hua Zhou (Ed.). International Joint Conferences on Artificial Intelligence Organization, 2549–2555. https://doi.org/10.24963/ijcai.2021/351 Main Track.

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV*.

Kirthevasan Kandasamy, Gur-Eyal Sela, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. 2020. Online learning demands in max-min fairness. *arXiv preprint arXiv:2012.08648* (2020).

G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 97–117.

Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks.

In *International Conference on Computer Aided Verification*. 443–452. https://doi.org/10.1007/978-3-030-25540-4_26

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems* 30 (2017). https://doi.org/10.5555/3295222.3295382

Haitham Khedr, James Ferlez, and Yasser Shoukry. 2020. PEREGRiNN: Penalized-Relaxation Greedy Neural Network Verifier. *arXiv preprint arXiv:2006.10864* (2020).

Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. International Conference on Learning Representations, (ICLR)*. OpenReview.net.

Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

Jingyue Lu and M Pawan Kumar. 2019. Neural network branching for neural network verification. *arXiv preprint arXiv:1912.01329* (2019).

Zhaoyang Lyu, Ching-Yun Ko, Zhifeng Kong, Ngai Wong, Dahua Lin, and Luca Daniel. 2020. Fastened crown: Tightened neural network robustness certificates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5037–5044.

Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288. https://doi.org/10.1145/3341302.3342080

Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2021b. Scaling Polyhedral Neural Network Verification on GPUs. *Proceedings of Machine Learning and Systems* 3 (2021).

Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2021a. Precise Multi-Neuron Abstractions for Neural Network Certification. *arXiv preprint arXiv:2103.03638* (2021).

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning Convolutional Neural Networks for Graphs. In *Proc. International Conference on Machine Learning, ICML*, Vol. 48. 2014–2023.

Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. 2021. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research* 59, 11 (2021), 3360–3377.

Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Semidefinite relaxations for certifying robustness to adversarial examples. *arXiv preprint arXiv:1811.01057* (2018).

Wonryong Ryou, Jiayu Chen, Mislav Balunovic, Gagandeep Singh, Andrei Dan, and Martin Vechev. 2021. Scalable Polyhedral Verification of Recurrent Neural Networks. In *International Conference on Computer Aided Verification*. Springer, 225–248.

Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. 2019. A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/246a3c5544feb054f3ea718f61adfa16-Paper.pdf

Ali Ghodsi Matei Zaharia Scott Shenker and Ion Stoica. 2013. Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints. (2013). https://doi.org/10.1145/2465351.2465387

Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. 2019a. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems* 32 (2019), 15098–15109.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018a. Fast and effective robustness certification. *Advances in Neural Information Processing Systems* 31 (2018), 10802–10813.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019b. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30. https://doi.org/10.1145/3290354

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019c. Boosting Robustness Certification of Neural Networks. In *International Conference on Learning Representations*.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM New York, NY, USA, 46–59.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018b. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.* 2, POPL (2018), 55:1–55:28.

Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. 2021. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3314–3320. https://doi.org/10.5555/3491440.3491898

Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. 2019. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 147–156. https://doi.org/10.1145/3302504.3311802

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

Christian Tjandraatmadja, Ross Anderson, Joey Huchette, Will Ma, KRUNAL KISHOR PATEL, and Juan Pablo Vielma. 2020. The Convex Relaxation Barrier, Revisited: Tightened Single-Neuron Relaxations for Neural Network Verification. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 21675–21686. https://proceedings.neurips.cc/paper/2020/file/f6c2a0c4b566bc99d596e58638e342b0-Paper.pdf

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=HyGIdiRqtm

Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T Johnson. 2020. Verification of deep convolutional neural networks using imagestars. In *International Conference on Computer Aided Verification*. Springer, 18–42.

Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Perfectly parallel fairness certification of neural networks. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428253

Joseph A Vincent and Mac Schwager. 2020. Reachable Polyhedral Marching (RPM): A Safety Verification Algorithm for Robotic Systems with Deep Neural Network Components. *arXiv preprint arXiv:2011.11609* (2020).

Binghui Wang, Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. 2021a. Certified robustness of graph neural networks against adversarial structural perturbation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1645–1653.

Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S. Yu. 2019. Heterogeneous Graph Matching Networks for Unknown Malware Detection. In *Proc. International Joint Conference on Artificial Intelligence, (IJCAI)*. 3762–3770.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018a. Efficient Formal Safety Analysis of Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 6369–6379. http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018b. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 1599–1614. https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. 2021b. Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *arXiv preprint arXiv:2103.06624* (2021).

Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. 2018. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*. PMLR, 5276–5285.

Eric Wong and Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*. PMLR, 5286–5295.

Haoze Wu, Clark Barrett, Mahmood Sharif, Nina Narodytska, and Gagandeep Singh. 2022a. *Artifact for Paper Scalable Verification of GNN- Based Job Schedulers*. https://doi.org/10.5281/zenodo.7080246

Haoze Wu, Alex Ozdemir, Aleksandar Zeljić, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. 2020a. Parallelization techniques for verifying neural networks. In *2020 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 128–137. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_20

Haoze Wu, Aleksandar Zeljić, Guy Katz, and Clark Barrett. 2022b. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 143–163.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020b. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. 2018. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems* 29, 11 (2018), 5777–5783.

Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. 2020. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *arXiv preprint arXiv:2011.13824* (2020).

Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. 2021. Improving neural network verification through spurious region guided refinement. *Tools and Algorithms for the Construction and Analysis of Systems* 12651 (2021), 389.

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proc. ACM SIGKDD Knowledge Discovery & Data Mining, KDD*. ACM, 974–983.

Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. 265–278. https://doi.org/10.1145/1755913.1755940

Tom Zelazny, Haoze Wu, Clark Barrett, and Guy Katz. 2022. On Optimizing Back-Substitution Methods for Neural Network Verification. *arXiv preprint arXiv:2208.07669* (2022).

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient Neural Network Robustness Certification with General Activation Functions. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/d04863f100d59b3eb688a11f95b0ae60-Paper.pdf