TOWARDS AN END-TO-END NEURAL-BASED FRAMEWORK FOR INTEGER OPTIMIZATION

by

Joachim Verschelde

Student number:	852594432	
Course code:	IM9502	
Thesis committee:	Prof Dr Stefano Bromuri (chairman),	Open University
	Murat Firat (supervisor),	Open University

1. INTRODUCTION

Linear programming and other mathematical optimization techniques, enable efficient solutions to tasks such as resource allocation, production scheduling, and routing.

Traditional solver-based approaches, suchs as the simplex method and interior-point method for linear programs and branch-and-bound-style approaches for mixed integer programming problems, typically rely on the problem structure to find a feasible solution. Each new problem instance is solved from scratch, which can be inefficient in contexts where similar instances recur or when decisions must be made in real-time Kotary et al. [2021]. In many practical scenarios (e.g. daily logistics with slowly changing data, or repeated simulations in a planning process), the type of problem instances share similar structural patterns. Recent work in Machine Learning integrates neural networks with optimization pipelines, because they offer the potential to learn the implicit structure of optimization problems from data Kotary et al. [2021] Wu and Lisser [2023] Nair et al. [2020] Fischetti and Jo [2018].

Broadly, two approaches are common.First, ML-augmented solvers integrate learned components into the traditional solving process Qian and Morris [2025]. For example, deep learning has been used to improve branch-and-bound decisions or cut selection in MILP solvers Nair et al. [2020], resulting in data-driven heuristics that complement existing algorithms. Second, end-to-end learning tries to directly predict the solution of an optimization problem using a neural network, essentially replacing the solver with a learned model Kotary et al. [2021] Tang et al. [2024] Lee and Kim [2024]. End-to-end approaches integrate easily into larger ML pipelines (since the solver is just a differentiable neural network). However, they also introduce significant challenges: how do we ensure that a neural network's output is feasible for a given problem instance? Most prior end-to-end methods struggle with these issues. Some require a large training set of solved instances (optimal solutions as labels) which is often expensive to obtain, while others resort to adding penalty terms for constraint violations or still rely on a conventional solver to repair or verify the network's solution Qian and Morris [2025]. These gaps motivate our work: we seek a learning-based optimization approach that is fast, self-improving, and aware of the underlying optimization structure, producing solutions that respect the problem's constraints.

In this paper, we propose an end-to-end neural-based framework for solving linear programs (LPs) and mixed-integer linear programs (MILPs) that addresses the above challenges. The core idea is to train a neural network to act as a solver: Given the description of an optimization instance, the network produces a candidate optimal solution. To guide the network toward valid solutions without requiring ground-truth labels for supervision, we incorporate the Karush-Kuhn-Tucker (KKT) conditions of optimality into the training objective.

The KKT conditions are a set of equations and inequalities that characterize optimal solutions to optimization problems. Under mild regularity conditions, any primal-dual solution satisfying KKT is guaranteed to be optimal. More information on the KKT-conditions can be found in the appendix A.1. By minimizing a loss function that measures violation of KKT conditions, we provide a signal to our network: the loss is zero if and only if the output represents an optimal solution for the given LP. This allows us to train the network

in an unsupervised way. The KKT objective residual becomes the loss signal so we have no need for explicit solution labels. Recent work has demonstrated the promise of this idea on simple LP examples Arvind et al. [2024], showing that training with KKT-based losses can outperform purely data-driven training in accuracy and generalization. Our framework extends this approach to MILPs. In addition to that, we will also replace the regular MLP from Arvind et al. [2024] and use Graph Neural Networks (GNNs) to represent the input optimization problems. In contrast with regular MLP networks, GNNs can work with inputs of different sizes, generalizing beyond fixed-size inputs.

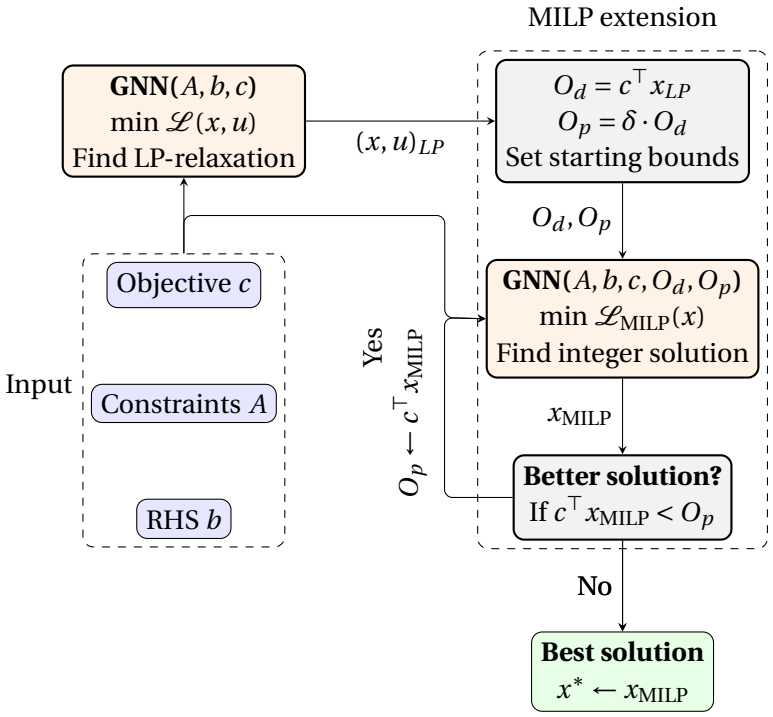

Figure 1: A conceptual pipeline for solving MILPs with two GNNs. First, *GNN*(A, b, c) solves the LP relaxation via a KKT-based loss, yielding a lower bound O_d. An upper bound is then set as $O_p = \delta \cdot O_d$, and *GNN*(A, b, c, O_d, O_p) is used to find a feasible integer solution by minimizing a MILP-specific loss. If a strictly better solution is found, O_p is updated and the second GNN is repeated.

In summary, our proposed framework combines deep learning techniques with mathematical optimization principles to solve LPs and MILPs in a novel way. Key contributions include the following:

- **Graph-based encoding of LP/MILP instances.** We use a bipartite GNN architecture Lee and Kim [2024] that embeds each instance's variable–constraint structure, giving the solver permutation invariance and the ability to generalize to problems of unseen size. This graph representation was not exploited by Arvind et al. [2024]
- **Extension for MILPs.** We extend the KKT-guided LP solver to MILPs by coupling two networks: one predicts a tight lower bound from the LP relaxation, the other projects to a feasible integer solution, iteratively tightening the upper bound.
- **Comprehensive empirical evaluation.** We plan a thorough empirical evaluation of our framework on optimization problems of different sizes using common benchmarks such as NETLIB for LPs and MIPLIB for MILPs.

The remainder of this proposal is structured as follows. In In Section 2, we provide background on neural networks for linear/mixed-integer programming (MLPs, GNNs). In Section 3, we review the relevant literature on integrating neural networks with LP and MILP solvers. Section 4 then formulates the specific research questions guiding our study. Next, Section 5 details our methodology and planning, including the timeline and milestones for implementation. Section 6 presents a risk analysis, enumerating potential challenges and mitigation strategies. Finally, the appendix contains supplementary material, such as extended proofs and implementation details.

2. Background

2.1. Multi-layer perceptrons

Forward-pass

Feedforward neural networks, or multi-layer perceptrons (MLPs), consist of layers of fully connected neurons that transform an input vector through weighted sums and nonlinear activation functions. Such networks require the input to be of a fixed size determined at training time. Consider follwing linear program in canonical form:

$$\begin{aligned} &\min_{x} c^\top x \\ \text{subject to} \quad &A x \leq b \end{aligned} \tag{1}$$

A straightforward approach would be to encode the objective coefficients c, constraint coefficients A and right-hand sides b as a single flat vector $\bar{x}$ and feed it into an MLP. Each neuron in the first layer will take a weighted sum of the input vector non-linear activation function to calculate the $\bar{x}$. The MLP would then be trained to predict the optimal objective value or decision variables (or an approximation). Let L be the number of hidden

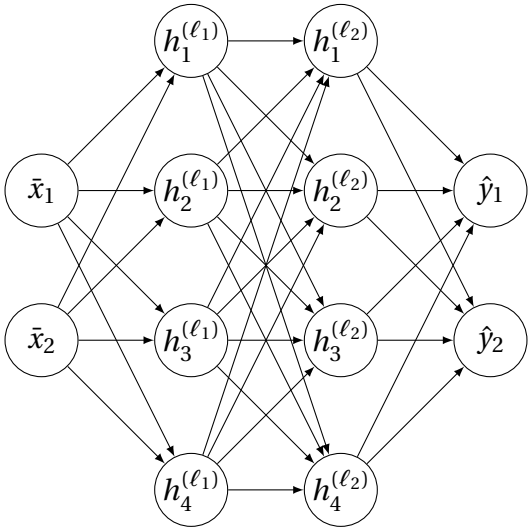

Figure 2: A simple feed-forward multilayer perceptron with 2 hidden layers.

layers and for convenience let $h^{(0)} = \bar{x}$. For every layer $\ell \in \{1, \ldots, L\}$ we have weight matrix $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ and bias $b^{(\ell)} \in \mathbb{R}^{d_\ell}$. The activation of layer ℓ is given by

$$h^{(\ell)} = \sigma(W^{(\ell)} h^{(\ell-1)} + b^{(\ell)}) \tag{2}$$

where σ is a non-linear activation such as RELU or tanh. For example the network's output is obtained in the following way:

$$\hat{y} = \sigma(W^{(\ell_3)} h^{(\ell_2)} + b^{(\ell_2)}) \tag{3}$$

BACKWARD-PASS

At the start of training, the weight matrices $W^{(\ell)}$ and bias vectors $b^{(\ell)}$ of each layer ℓ are randomly initialized. Given a training set of N examples $\{(\bar{x}^{(1)}, y^{(1)}), \ldots, (\bar{x}^{(N)}, y^{(N)})\}$ we seek to learn the parameters by minimizing an average loss:

$$\min_{W,b} \mathscr{L}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} L\big(\hat{y}^{(i)}, y^{(i)}\big), \tag{4}$$

where $L(\cdot,\cdot)$ is a suitable loss function, such as mean-squared error or cross-entropy. To optimize the network parameters, we employ gradient descent. The key step is to compute the gradient of the loss $\mathscr{L}$ with respect to each parameter $W^{(\ell)}$ and $b^{(\ell)}$, working backwards from the output layer to the input layer via back-propagation. This procedure applies the chain rule layer by layer; for each $\ell \in \{L, \ldots, 1\}$

$$W^{(\ell)} \leftarrow W^{(\ell)} - \alpha \frac{\partial \mathscr{L}}{\partial W^{(\ell)}}, \quad b^{(\ell)} \leftarrow b^{(\ell)} - \alpha \frac{\partial \mathscr{L}}{\partial b^{(\ell)}},$$

where α is the learning rate, and $\frac{\partial \mathscr{L}}{\partial W^{(\ell)}}$, $\frac{\partial \mathscr{L}}{\partial b^{(\ell)}}$ are obtained by successively applying the chain rule through all subsequent layers. By iterating this update procedure multiple times, the network parameters are gradually refined so that the network's predictions $\hat{y}$ better fit the solutions to the problem instances.

2.2. GRAPH NEURAL NETWORKS

Graph Neural Networks offer a powerful way to encode optimization instances of varying sizes. Instead of flattening problem data into a single fixed-length vector (as with MLPs), GNNs represent each instance as a graph whose nodes and edges reflect the problem's structure as can be seen in figure 3. A typical example in the MILP context is the bipartite graph representation, which treats decision variables and constraints as two distinct sets of nodes Nair et al. [2020] Lee and Kim [2024].

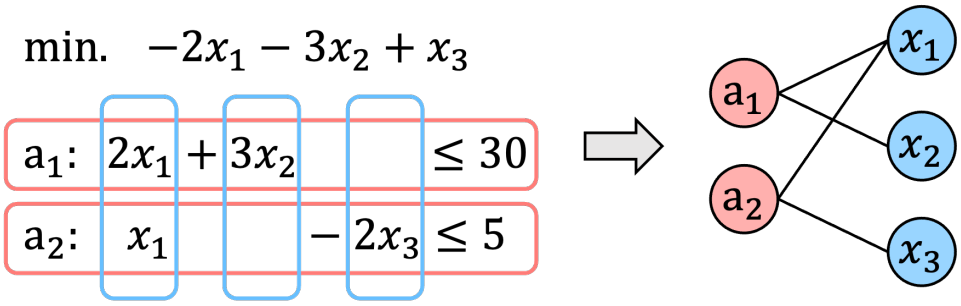

Figure 3: A small MILP (left) with its coefficient matrix A in the middle. Rows (constraints) become red nodes a_1, a_2, while columns (variables) become blue nodes x_1, x_2, x_3. Edges represent nonzero A_{ij}.

BIPARTITE GNN ARCHITECTURE

To construct the bipartite graph, one set of nodes corresponds to the constraints (rows of A) and the other set corresponds to the decision variables (columns). An edge exists between a constraint node and a variable node if and only if the corresponding coefficient in A is nonzero. Each node then holds features (e.g., objective coefficients c_j, right-hand sides b_i, or bounds), and during message passing, constraints pass information to variables and variables pass information back to constraints. This two-way flow of information helps the network learn to minimize violations, guide feasibility, and push solutions toward optimality.

PERMUTATION INVARIANCE

Since GNNs process nodes based on their connectivity rather than relying on a specific node ordering, the architecture is invariant to permutations. This is especially useful for MILPs, where the labeling or order of constraints and variables should not affect the solution. Whereas MLP inputs must be ordered in some way (and shuffling the input could produce different outputs), GNNs aggregate messages from neighbors in a symmetric fashion (e.g., summation or averaging), ensuring the model's output remains consistent regardless of how the problem is indexed.

3. RELATED WORK

3.1. LINEAR PROGRAMS

Wu and Lisser [2023] used the KKT conditions explicitly. The KKT conditions are employed as (boundary) conditions for a first-order ODE (dynamic system). The idea is that this dynamic system converges, as $t \to \infty$, to a state that satisfies the KKT conditions. It is claimed that the solution can thus be found. However, for each LP, a new neural network must be trained, which makes the method very slow and computationally intensive.

Chen et al. [2022] have modeled an LP as a bipartite GNN. Their goal is to generally study solution methods for LPs via bipartite GNNs. However, no concrete guidelines are provided on how this GNN should be trained exactly. It also remains unclear how the objective values and solutions can be reliably determined from the network. GNNs are illustrated for LPs with 10 variables and 50 inequalities, which then generate "feasibility", objective value, and a solution. The GNNs are trained via supervised learning, with the solver SCIP serving as ground truth.

Qian et al. [2024] provide an explanation (interpretation) of how GNNs can be used to solve LPs. Concretely, the Message Passing steps in the GNN are viewed as the steps of an interior-point algorithm. The GNN is trained with supervised learning, based on solutions generated by a solver (using an interior-point method).

3.2. MIXED INTEGER LINEAR PROGRAMS

Scavuzzo et al. [2024] reviewed the current literature on the use of Machine Learning for MILP. In it, several approaches are discussed and different methods are compared. They broadly distinguish:

1. Primal heuristics
2. Branch and Bound

3. Cutting planes

Primal heuristics

A GNN is used here to solve a (sub)part of the problem, corresponding to "stable" variables. The subproblem with the remaining variables is then solved by a solver (usually SCIP). It has been established that for many binary MILPs, in numerous feasible solutions, certain variables consistently take the same values in all (or in most) feasible solutions. These are called stable variables. Several GNNs have thus been built to predict these stable variables, after which the other variables in a subproblem are determined by a solver.
Ding et al. [2020] trained a GNN with attention via supervised learning to predict the stable variables. A (binary) cross-entropy is used as the loss function. A local branching step in SCIP is used to determine the remaining variables.
Nair et al. [2020] introduced a novel GNN architecture consisting of two components: Neural Diving and Neural Branching. The Neural Diving is a primal heuristic similar to that of Ding et al. [2020], but they use a different loss function, in which the binary label probability (the probability distribution over the stable variables) is adjusted.

We have seen in the literature that there are GNN methods that, for example, train on interior-point solutions (cf. Qian et al. [2024]) or use KKT conditions (cf. Wu and Lisser [2023]), but these are relatively slow. The target probability distribution is given by:

$$P_T\left(x^k\right) = \frac{\exp\left(-c^\top x^k\right)}{\sum_{i=1}^{K} \exp\left(-c^\top x^i\right)}, \tag{5}$$

where $\{x^1, x^2, \ldots, x^K\}$ is a set of K feasible solutions.
This way, feasible solutions are generated near a (potentially) lower objective value. The Neural Branching GNN aims to predict which variables will undergo Full Strong Branching, so that the root search tree remains as small as possible. Imitation learning is used here, following the expert policy from SCIP.
Khalil et al. [2022]'s approach also uses a GNN and closely aligns with the work of Ding et al. [2020] As the target probability distribution, the average value of the binary variables is used:

$$P_T(x_j = 1) = \frac{1}{K} \sum_{k=1}^{K} x_j^k. \tag{6}$$

These approaches are only semi end-to-end in the sense that part of the MILP is determined by the neural network, but a solver is still needed to complete the solution.

Branch and Bound

These approaches are not end-to-end at all and only assist the solver in making good decisions during the branch-and-bound algorithm. Gasse et al. [2019] use a bipartite GNN to help solve MILPs. The GNN predicts the variable on which to branch. Learning is done via imitation learning, using a solver (SCIP) to obtain the "correct" branching variables as examples.
Zarpellon et al. [2021] do not use a representation of the MILP itself (A, B, c) but rather of the B&B search tree. The authors state that this approach generalizes better beyond the

trained classes of combinatorial optimization problems. The expert used is SCIP's hybrid-branching expert.
Lin et al. [2022] continue in the same vein as Zarpellon et al. [2021]: branching decisions are predicted based on the description of the branch-and-bound search tree and thus not directly on the MILP parameters. The neural network used is a transformer, and multitask learning is employed, with SCIP as the expert.

CUTTING PLANES

Cutting-plane methods are part of modern MILP solvers. In so-called separation rounds, cuts are selected according to certain criteria and added to the original MILP so that the previous LP is no longer feasible.

Since a cutting-plane method as described bove is not used in ML for cut selection (which is usually based on imitation learning with a solver as the expert), and since a cutting-plane method as a standalone method to solve MILPs is not feasible, the literature on this topic is not really relevant to our goal of developing an end-to-end ML method for MILPs.

3.3. END TO END METHODS

End-to-end ML methods for solving MILPs are very recent. The method of Tang et al. [2024] is quite elegant and consists of a NN that solves the relaxation LP $\rightarrow$ followed by a constraint layer that projects the constructed solution onto integer values. The entire network is trained according to a loss function that aims to minimize the objective function as much as possible, while imposing a penalty for violation of the inequalities:

$$\mathscr{L} = \frac{1}{m}\sum_{i=1}^{m}\Big(f(x^i,\xi^i)\Big) + \lambda\Big\|\mathrm{ReLU}\big(g(x^i,\xi^i)\big)\Big\|^2 \tag{7}$$

where f and g are the objective function and the inequality constraints, respectively, which are not necessarily linear. The NN is a network with 5 fully connected layers, whose width is adapted to the size of the MILP. Thus, the NN does not really generalize to larger MILPs.

Lee and Kim [2024] use RL to solve a MILP. The action $A_t = (a_{t,1}, a_{t,2}, \ldots, a_{t,m})$ of the m integer variables, where $a_{t,i}(\pm 1)$, corresponds to the decision whether the i-th variable remains the same or is increased or decreased by 1. It is clear that a feasible solution must first be found and only then can that feasible solution be optimized. This system therefore works in two phases. In the first phase, a positive reward is given if the number of inequality violations is reduced, regardless of the objective value, until a good feasible solution is reached. In the second phase, improving the objective value is rewarded, whereas if inequality violations occur again, reducing the number of violations is once again rewarded. The NN used is a GNN.
Donti et al. [2021] introduce a Lagrangian penalty method for imposing feasibility. The method applies to general non-linear problems, and therefore also to LPs. However, feasibility (satisfying the inequalities) is limited to real-valued variables. The method is unsupervised and is based on the following loss function:

$$\mathscr{L} = f(x) + \lambda\left\|\mathrm{ReLU}\big(g^*(y)\big)\right\|^2 \tag{8}$$

where f is the objective function and g the inequalities. This NN is a fully connected network with 2 hidden layers of width 200. Hence, this NN does not generalize to much larger instances either.

3.4. Data set generation

Generating feasible LPs or MILPs is a non-trivial problem. Many papers rely on the MILP form of CO problems that are relaxed for LPs, thereby ensuring feasibility. For example, in the paper of Qian et al. [2024], instances for 4 types of CO problems are generated: the set cover problem, the maximum independent set problem, the combinatorial auction problem, and the capacitated facility location problem. This yields a broad class of MILPs/LPs. Likewise, Nair et al. [2020] use MILPs from data sources such as MIPLIB and Google Production Planning. However, if one requires an even broader range of MILPs as a dataset, another approach is needed. For example Wang et al. [2023] or Zhang et al. [2024] [TODO describe papers]
Similarly, Li et al. [2024] introduce MILP-Evolve, an LLM-based data-generation pipeline that produces various MILP blocks and instances from a limited set of seed-MILP combinations through prompting, tailored specifically to the MILP domain. By adjusting and refining prompts, the LLM can create new and diverse MILP blocks that deviate significantly from the original examples. By basing the prompts on realistic scenarios and industrial applications, the relevance and diversity of the generated MILP blocks is increased. GPT-4 is used as the LLM. Since our intention is to focus on developing a new end-to-end method (based on a new self-supervised/unsupervised learning approach with a new loss function), we will limit ourselves dataset-wise to these new Qian et al. or Nair et al. We will not delve further into these new developments in MILP instance generation.

4. Research Questions

4.1. Encoding the KKT Conditions of a Linear Program via a GNN architecture

The first research question is the following:

> **RQ1: Can we model the KKT-conditions of a linear program (LP) as a minimization problem using a graph neural network architecture?**

In a classical linear program defined by:

$$\min_x c^\top x \quad \text{subject to} \quad Ax \le b,$$

the Karush–Kuhn–Tucker (KKT) conditions Kuhn and Tucker [1951] introduce Lagrange multipliers $u \ge 0$ and state that optimal solutions (x^*, u^*) must satisfy

$$\begin{cases} A^\top u + c = 0, \\ Ax - b \le 0, \\ (Ax-b)_i \, u_i = 0 \quad \forall\, i, \\ u_i \ge 0 \quad \forall\, i. \end{cases} \tag{9}$$

One way to remove explicit inequalities is to use the ReLU function first introduced by Fukushima [1975]:

$$\mathrm{ReLU}(z) = \max(0, z).$$

The main intuïtion is that $\text{ReLU}(Ax-b)=0 \iff Ax-b\leq 0$. Furthermore, we can rewrite $u_i \geq 0 \quad \forall\, i$ as $\text{ReLU}(-u_i)=0$. This way, we can rewrite all KKT conditions as equalities. To encode these KKT conditions as a minimization problem, we propose a positive objective:

$$\mathscr{L}(x,u)=\sum_{i=1}^{n}(A^{\top}u+c)_i^2+\sum_{j=1}^{m}\text{ReLU}\big((Ax-b)_j\big)+\sum_{j=1}^{m}\text{ReLU}(-u_j)+\sum_{j=1}^{m}u_j^2(Ax-b)_j^2 \tag{10}$$

Each term is nonnegative and vanishes only when the respective KKT condition is exactly satisfied. It is easy to show that this objective has no local minima and is exactly zero at the optimal solution. Therefore, if a neural network is designed to generate (x,u) and the above function is used as a loss, then $\nabla\mathscr{L}(x,u)=0$ forces the solution to meet all KKT conditions. A proof for this is included in the appendix A.3. More specifically we want to:

- Demonstrate that we can design a graph neural network to output feasible (x,u) that minimize $\mathscr{L}(x,u)$ for optimization problems of different sizes.
- Compare the solver performance (both in accuracy and runtime) to classical LP solvers.

4.2. Moving to MILP - Reusing the Same Architecture

The second research question investigates if we can reuse the architecture from RQ1 for MILPs:

> **RQ2: Can we reuse the architecture from RQ1 for mixed-integer linear programs? If so, what modifications are required?**

A mixed-integer linear program has the form:

$$\min_x c^{\top}x \quad \text{subject to} \quad Ax\leq b, \quad x_i\in\{0,1\} \text{ or } \mathbb{Z}, \tag{11}$$

where a subset (or all) of the decision variables are constrained to be integral. Although one may still use the KKT-based approach for the continuous relaxation, integrality constraints add complexity. Naively, a continuous relaxation will not enforce $x_i\in\{0,1\}$; we need additional terms in the objective (or specialized layers) to push x_i to integers.

One popular penalty for enforcing binary solutions is:

$$\sum_{i=1}^{n}x_i^2(1-x_i)^2,$$

which vanishes only if x_i is exactly 0 or 1. We can then combine this term with the KKT-based penalty for the continuous portion. However, MILPs are mostly solved using a branch-and-bound style search, where feasible integral solutions provide improved upper bounds, and one prunes regions that cannot yield better solutions. Our approach will be the following:

1. Solve the LP relaxation of the MILP.
2. Use the resulting continuous solution to derive a lower bound O_d.
3. Impose an upper bound $O_p=\delta\cdot O_d$ that guides the network to feasible integer points with $O_d\leq c^{\top}x\leq O_p$.

4. Iterate and refine, lowering O_p whenever we discover a better feasible integer solution.

Algorithm 1 Iterative MILP Algorithm

Input: MILP instance (A, b, c), an LP-relaxation solution x_{LP} and a parameter δ to expand the search interval.
$O_d \leftarrow c^T x_{\text{LP}}$
$O_p \leftarrow O_d\,(1+\delta)$
$x_{\text{old}} \leftarrow x_{\text{LP}}$
while true **do**
    $x_{\text{new}} \leftarrow \text{GNN}(A, b, c, O_d, O_p)$ ▷ Obtain a candidate solution within $[O_d, O_p]$
    $O_p \leftarrow c^T x_{\text{new}}$
    if $\dfrac{c^T x_{\text{new}} - c^T x_{\text{old}}}{c^T x_{\text{old}}} > \epsilon$ **then**
        $x_{\text{old}} \leftarrow x_{\text{new}}$
    else
        break ▷ Stop if improvement is below threshold
    end if
end while
$x^* \leftarrow x_{\text{new}}$ ▷ Final solution

GNN (A, b, c, O_d, O_p) refers to a (graph neural-network-based) procedure or model whose feasible solutions x satisfy

$$\begin{aligned} A\,x - b &\le 0, \\ x_n &\in \{0,1\}, \quad \text{for each integer index } n, \\ O_d &\le c^T x \le O_p, \end{aligned} \tag{12}$$

where

$$O_d = \text{dual bound}, \quad O_p = \text{primal bound}. \tag{13}$$

A suitable loss function for driving a neural network toward such feasible x could be

$$\mathscr{L}_{\text{MILP}}(x) = \sum_{j=1}^{m} \text{ReLU}\big((Ax-b)_j\big) + \text{ReLU}\big(-c^{\top}x + O_d\big) + \text{ReLU}\big(c^{\top}x - O_p\big) + \sum_{i=1}^{n} x_i^2\,(1-x_i)^2 \tag{14}$$

When x is truly feasible and $c^T x$ lies between O_d and O_p, the above terms vanish and $\mathscr{L}_{\text{MILP}}(x)$ can reach 0.

A more formal explanation for this system can be found in the appendix A.4. RQ2 explores whether the same architecture can be extended straightforwardly to the discrete case by adding an integrality penalty and a bounding strategy.

4.3. Reusing Infeasible Solutions to Guide the Search

Another interesting question is:

> **RQ3: Can infeasible solutions be reused in order to guide the search towards feasible solutions?**

In regular optimization solvers, an infeasible solution indicates a dead end, either the solver backtracks or adds a cut. In our neural-based approach, however, an infeasible solution might carry important gradient information. For example, large constraint violations could produce strong gradients that can guide how to adjust x in a subsequent iteration. Thus, rather than discarding infeasible iterates, we may:

- Store them in a experience replay buffer Lin [1992].
- Re-initialize the network from an infeasible point to see if small changes can make it feasible.
- Use them to weight certain constraints that are violated more frequently, which in turn increases feasibility.

Here we examine whether reuse of infeasible solutions can improve convergence rates, reduce the number of infeasible attempts, or yield better final solutions. The idea for this comes from reinforcement learning where prioritized replay buffers contain experiences that yield a larger error are given higher sampling probability Schaul et al. [2015].

5. Planning

5.1. Overview

This planning outlines the step-by-step process for our research project, running from the start of April until the end of September which is a total of 26 weeks. The primary objective is to investigate how to encode the KKT conditions for LPs and then adapt the same framework for MILPs. We also explore how infeasible solutions can guide improved optimization performance. The plan is divided into four phases and a final wrap-up.

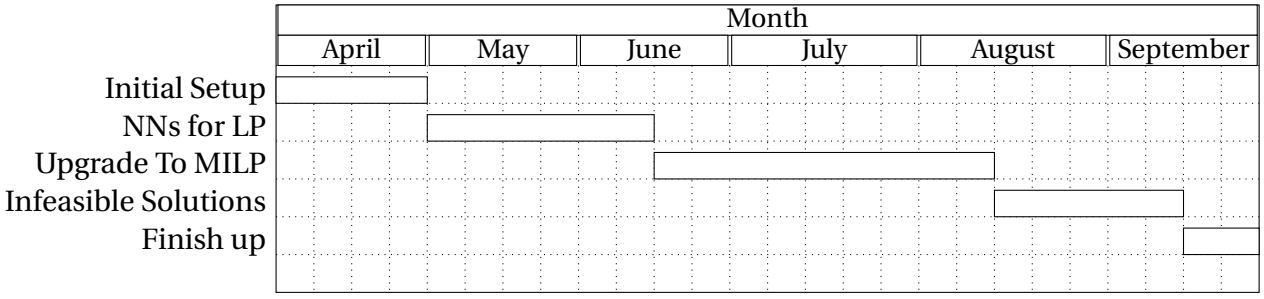

Figure 4: Gantt chart visualising the planning outline for the research project

5.2. Initial Setup

Set up the environment for a neural network architecture

In early April, we will prepare the software and hardware environment required for development. This includes setting up a docker container with a certain base image containing Python, NumPy, PyTorch, SciPy, SCIP and the CUDA toolkit needed for accelerated training.

Look for a third-party Python dataloader for the NETLIB LP benchmark

Simultaneously, we will search for an existing Python library to load small- to medium-scale linear programs from the NETLIB benchmark. If no sufficiently robust loader exists, we will implement a minimal dataloader ourself.

Set up the existing GNN from GitHub

We will spend the next 2 weeks exploring existing GNN architectures such as `https://github.com/chendiqian/IPM_MPNN`. Further more we will need to setup the dependencies for the GNN repository in our docker container. Other tasks may be the exploration of the GNN layers and see how the data flows from input to output or plan how to integrate our custom loss function.

Further define evaluation metrics

Finally, we will determine how to measure our solver's performance and compare it to a baseline solver like SCIP. Possible metrics could be:

- Feasibility rate
- Optimality gap
- Runtime performance
- Scalability with respect to problem size

By the end of Week 4, the environment will be set, a NETLIB loader in place, the GNN cloned and running, and all metrics clearly defined.

5.3. Create Network Architecture for LPs

First variant of the GNN with custom loss $\mathscr{L}(x, u)$

In Weeks 5–7, we will develop our initial network to handle LPs. Our custom loss function that encodes the KKT conditions 10. This means implementing the loss function $\mathscr{L}(x, u)$ within the GNN training loop. Next we need to verify if the gradients are computed correctly and the training is stable. Finally we may need to adjust GNN layers or hyperparameters (batch size, learning rate, etc.).

Solve the LP problems using SCIP

While building and testing the network, we will also run SCIP on each NETLIB instance over approximately two weeks. This provides a reliable baseline and the results will serve as ground truth when comparing our neural approach.

Record performance

By Week 10 we will generate tables and plots showing:

- Objective gaps relative to SCIP solutions
- Feasibility rates
- Time-to-convergence comparisons

The results will indicate whether our neural approach is competitive in both accuracy and runtime, providing insights on required refinements.

5.4. Upgrade Architecture for MILP

Second NN with custom function $\mathscr{L}_{\text{MILP}}(x)$

Next, we shift focus to MILPs and we extend our architecture for handling integrality constraints using a second NN with our updated loss function.

Search for a third-party Python dataloader for MILPLIB

In parallel, we will look for or adapt an existing loader for MILPLIB. The goal is to replicate our NETLIB workflow but for MILP instances.

Implement a custom optimization loop

Over the next three weeks, we will code a our training loop that:

- Runs our first network to get x^*_{LP} and sets $O_d = c^\top x^*_{LP}$.
- Initializes $O_p = \delta \times O_d$ for some $\delta > 1$.
- Trains the MILP network to find integer-feasible solutions with $c^\top x \leq O_p$.
- Iteratively lowers O_p whenever an improved solution is found.

Solve the MILPs using SCIP

We will also solve the MILPs from MILPLIB using SCIP. This step should confirm whether our approach competes with standard MILP solvers.

Record performance

For the next week we will compare results from our approach versus SCIP. Important observations could be:

- Feasibility of integer solutions
- Quality of the objective values found
- Overall runtime

By Week 19, we should have a clear sense of how well the KKT-inspired architecture generalizes to the MILP setting.

Write paper

We also plan to start drafting our paper around Week 19 and bundeling the results that we already collected.

5.5. Learn from Infeasible Solutions

Implement a prioritized replay buffer

Addressing the question of reusing infeasible solutions, we will implement a replay mechanism where solutions with large constraint violations or errors are sampled more often during training. This approach, borrowed from reinforcement learning, can push the network to learn corrections for typical pitfalls more rapidly.

Update the optimization loop and training pipeline

Next, we will integrate this buffer into our existing LP and MILP loops. We will:

- Retain infeasible samples instead of discarding them.
- Assign sampling priorities based on the magnitude of constraint violations.
- Regularly sample from the buffer and update the network's parameters.

Compare performance with or without reusing infeasible solutions

We will run controlled experiments, tracking:

- Final solution quality
- Training stability
- Runtime efficiency

We want to see if reintroducing infeasible samples does indeed reduce the overall number of iterations needed or improves the final objective.

Write results and evaluation

Finally, we will summarize our findings in a short write-up to be integrated into the main paper. We will highlight any improvements in convergence speed or solution quality attributed to infeasible solution reuse.

5.6. Finish Up

Finalizing the paper

The last two weeks of September will be devoted to finishing the final paper. We will:

- Merge the sections from previous phases.
- Create consistent tables and figures.
- Write conclusions and outline possible future work.

The final result should address our three research questions, offering a end to end KKT-based neural approach for LPs, a generalized version for MILPs, and an analysis of how infeasible solutions can guide the training process.

6. Risk Analysis

In this section, we examine a range of potential risks associated with our proposal. Our approach carries certain risks that must be clearly identified and mitigated. Several challenges can arise in the course of integrating neural network approaches with optimization pipelines. These include concerns about computational overhead, convergence reliability, generalizability to new problem instances, data quality and diversity and much more. Additionally, bridging theoretical insights from convex optimization with practical neural-network training may introduces unforeseen complexities. Below, we detail ten key risks, each of which could affect the success of our methodology. We first discuss each risk qualitatively, noting how it might arise and why it can adversely impact our project. Table 1 summarizes these risks in three columns, indicating their gravity, a concise description, and a proposed mitigation strategy.

Gravity	Description	Mitigation
High	KKT-based losses for large-scale LP/MILPs can become memory-intensive and lead to slow training times.	Start with smaller benchmark instances (NETLIB/MIPLIB) and scale up gradually, employ hardware acceleration.
High	Introducing a custom loss function may invalidate standard hyperparameters, causing instability or slow convergence.	Perform hyperparameter tuning (e.g., Optuna), start with recommended GNN defaults and closely monitor training curves and loss decomposition to diagnose issues.
Medium	Relying on narrow benchmark repositories can lead to biased models that fail on diverse real-world distributions.	Use multiple repositories (NETLIB, MIPLIB, etc.) and, if needed, synthetic data generation (e.g., LLM-based).
Medium	Resource-heavy GNN architectures and iterative gradient-based training might exhaust CPU/GPU budgets, delaying progress.	Begin with small instances to estimate resource usage; scale up iteratively. If needed, focus on a representative subset of problem instances to reduce computational load.
Medium	Bugs or lack of support in libraries (e.g., SCIP, GNN repos) can stall progress or force last-minute migrations.	Containerize with Docker and pin library versions.
Medium	Multiple redesigns and experiments (e.g., architecture changes, debugging) can exceed planned schedules.	Use incremental milestones; maintain regular progress reviews, start with a minimal viable approach on small LPs, then extend to MILPs.
Low	Undocumented neural methods or hyperparameters can hinder independent verification of results.	Open-source all code and environment configurations document seeds, training protocols, and data pipelines

Table 1: Updated overview of key risks, their gravity, brief descriptions, and proposed mitigation strategies.

6.1. Complexity of the KKT-Based Loss for Large-Scale Problems.

Our KKT-based loss function - which encodes constraints, dual feasibility, and complementary slackness into a single differentiable objective - can become computationally expensive for large-scale LP or MILP instances. While each component of the loss is straightforward to compute for small systems (as in Fischetti and Jo [2018]; Wu and Lisser [2023]), scaling up to hundreds of thousands of variables or constraints may demand a lot of memory and long training times.

Mitigation

We will start with smaller, controlled instances from well-known benchmarks (NETLIB for LPs, MIPLIB for MILPs) and gradually scale up in problem size. We will use hardware acceleration to handle large-scale settings.

6.2. Problems with predefined hyperparameters

Because we introduce a custom loss function, we should be aware that hyperparameters that would work for a predefined GNN-approach may not work anymore and may need finetuning.

Mitigation

We will start with the recommended hyperparameters and use cross-validation with a hyperparemeter-tuning framework such as Optuna to efficiently search the hyperparameter space. We will monitor training curves and loss decomposition to diagnose potential problems.

6.3. Data Generation and Bias

A common pitfall is reliying a limited or biased set of benchmark problems. While standard repositories (NETLIB, MIPLIB) are invaluable, they may not reflect the diverse real-world distributions that an end-to-end solver could face.

Mitigation

We will use diverse sources of training and testing data, drawing from multiple benchmarks and repositories. If needed, we can adopt instance generation methods such as LLM-based scenario creation. Li et al. [2024]

6.4. Insufficient Computational Resources jeopardizing the timeline

Because we will rely on (possibly large) GNN architectures, iterative gradient-based training, and hyperparameter-finetuning. This approach could exhaust available CPU/GPU resources, especially with large problem instances. This in turn could jeopardize the project timeline.

Mitigation

We will start with small problem instances and scale iteratively to large instances. This allows us to estimate memory consumption and time complexity. Should resources become a bottleneck, we will prioritize a subset of representative instances rather than aim for exhaustive coverage.

6.5. Dependency on Third-Party Libraries or Tools

Our research depends on external libraries such as SCIP, data loaders for NETLIB/MIPLIB, or a GNN-model from an existing GitHub repository. If there are problems with these libraries such as undiscovered bugs, or insufficient support, our project might be delayed or require an alternative solution.

Mitigation

We will containerize our environment with Docker or a similar technology, pinning specific library versions to ensure reproducibility.

6.6. Uncertain Timeline Due to Iterative Development

Developing a new neural approach for optimization will involve multiple iterations of architecture design, hyperparameter tuning, debugging, and re-running experiments. This iterative nature can exceed initial time estimates. Each cycle might uncover new technical challenges or require design changes that propagate through our pipeline, again leading to pressure on the timeline.

Mitigation

We will consitently follow up on our planning (see Section 5). We will hold regular progress meetings, adjusting the scope if we find that certain tasks consistently slip. Starting with a minimal viable system that can already solve small LP. And moving to larger MILP instances and more advanced features such as the replay buffer for infeasible solutions.

6.7. Reproducibility

Neural methods can sometimes be opaque. If our code and data pipelines are not openly available, or if random seeds and hyperparameters are not documented, independent verification might fail.

Mitigation

We will release our source code in a public GitHub repository under an appropriate license, carefully documenting environment setups, dependencies, seeds, and training protocols.

A. Problem Analysis

A.1. Linear Programming and KKT

We define a linear program as follows:

$$\min_x c^\top x \quad \text{s.t.} \quad Ax \le b \tag{15}$$

where

$$\begin{aligned} x &= (x_1, \ldots, x_n) \\ c &= (c_1, \ldots, c_n) \\ b &= (b_1, \ldots, b_m) \end{aligned} \tag{16}$$

We introduce Lagrange multipliers $u = (u_1, \ldots, u_m)$ with $u \ge 0$ and defined the Lagrangian as:

$$L(x,u) = c^\top x + u^\top \big(Ax - b\big) \tag{17}$$

From a saddle-point or min–max perspective,

$$\max_{u \ge 0} L(x,u) = \begin{cases} c^\top x, & \text{if } Ax \le b, \\ +\infty, & \text{otherwise} \end{cases} \tag{18}$$

we noted that minimizing $c^\top x$ subject to $Ax \le b$ is equivalent to

$$\min_x \max_{u \ge 0} L(x,u) \tag{19}$$

where

$$L(x,u) = c^\top x + u^\top \big(Ax - b\big), \quad u \ge 0 \tag{20}$$

The solution (x^*, u^*) of the LP must therefore be a minimum in x and a maximum in $u \ge 0$ for $L(x,u)$.

$$\begin{aligned} \min_x L(x,u) = \frac{\partial L}{\partial x} &= 0 \\ &= c^\top + u^\top A = 0 \\ &= A^\top u + c = 0 \end{aligned} \tag{21}$$

$$\max_{u \ge 0} L(x,u) = \max_{u \ge 0} \big[u^\top (Ax - b) \big] \tag{22}$$

since x is treated as a constant when we vary u. We then consider each component $(Ax - b)_i$:

- If $(Ax-b)_i < 0$, then with $u_i \ge 0$, the product $u_i (Ax-b)_i$ is largest if $u_i = 0$.
- If $(Ax-b)_i = 0$, then $u_i \ge 0$ can be arbitrary, because $u_i 0 = 0$.
- If $(Ax-b)_i > 0$, then letting $u_i \to \infty$ would make $u_i (Ax-b)_i$ unbounded (i.e. the objective goes to $+\infty$).

Hence, for a finite optimum to exist, we must have

$$(Ax-b)_i \le 0 \quad \text{for all } i, \tag{23}$$

and if $(Ax-b)_i = 0$, then u_i may be positive, whereas if $(Ax-b)_i < 0$, we must take $u_i = 0$. Summarizing the conditions for a single (finite) solution in x and u:

$$\begin{aligned} A^\top u + c &= 0, \\ u_i = 0 &\iff (Ax-b)_i < 0, \\ u_i \ge 0 &\iff (Ax-b)_i = 0 \end{aligned} \tag{24}$$

Equivalently, we can write:

$$(Ax-b)_i \le 0, \qquad (Ax-b)_i\, u_i = 0, \qquad u_i \ge 0, \quad \forall\, i \tag{25}$$

Hence, solving the linear program from 15 is equivalent to finding x and $u \ge 0$ satisfying the KKT conditions Kuhn and Tucker [1951]:

$$\begin{cases} A^\top u + c = 0, \\ Ax - b \le 0, \\ (Ax-b)_i\, u_i = 0 \quad \forall\, i, \\ u_i \ge 0 \quad \forall\, i. \end{cases} \tag{26}$$

A.2. Eliminating Inequalities via ReLU

We now show how to remove the explicit inequalities by introducing the "ReLU" function:

$$\mathrm{ReLU}(z) = \max(0, z). \tag{27}$$

We then see that:

$$\mathrm{ReLU}(Ax-b) = 0 \iff Ax - b \le 0 \tag{28}$$

Applying this idea to the KKT conditions, recall that for each i we need:

$$\begin{aligned} (Ax-b)_i &\le 0, \\ u_i &\ge 0, \\ (Ax-b)_i\, u_i &= 0 \end{aligned} \tag{29}$$

The last equation, known as complementary slackness, says that either $(Ax-b)_i = 0$ or $u_i = 0$. A convenient way to encode these constraints simultaneously is:

$$\mathrm{ReLU}\big(u_i + (Ax-b)_i\big) = u_i \tag{30}$$

When $(Ax-b)_i \le 0$ and $u_i \ge 0$ with $(Ax-b)_i\, u_i = 0$, one can check that this identity indeed holds. In effect, the ReLU-based formulation enforces nonnegativity, the inequality constraint, and complementary slackness all at once.

Hence, the KKT system without explicit inequalities can be written as

$$\begin{cases} A^\top u = c, \\ \mathrm{ReLU}\big(u + Ax - b\big) = u. \end{cases} \tag{31}$$

This removes the need to write out $Ax - b \le 0$, $u \ge 0$, and $(Ax-b)_j\, u_j = 0$ explicitly, since the ReLU formulation encodes them automatically.

A.3. Reformulation of Our LP and KKT conditions as minimizing a nonnegative Function

We want to reformulate the linear program from 15 and its KKT conditions from 26 as the minimization of a single, unconstrained nonnegative function.

$$\begin{aligned}\mathscr{L}(x,u) &= \sum_{i=1}^{n}(A^\top u + c)_i^2 + \sum_{j=1}^{m}\mathrm{ReLU}\big((Ax-b)_j\big) + \sum_{j=1}^{m}\mathrm{ReLU}(-u_j) + \sum_{j=1}^{m}u_j^2(Ax-b)_j^2 \\ &= \sum_{i=1}^{n}(c_i + \sum_{J=1}^{m}A_{ji}u_j)^2 + \sum_{j=1}^{m}\mathrm{ReLU}(\sum_{i=1}^{n}A_{ij}x_j - b_j) + \sum_{j=1}^{m}\mathrm{ReLU}(-u_j) + \sum_{j=1}^{m}u_j^2(\sum_{i=1}^{n}A_{ij}x_i - b_j)^2\end{aligned} \tag{32}$$

$\mathscr{L}(x,u)$ is a sum of positive terms:

$$\mathscr{L}(x,u) = \sum_{k=1}^{K} f_k(x,u) \quad \text{with} \quad f_k(x,u) \geq 0 \quad \forall (x,u) \tag{33}$$

For a local minimum (x_ℓ, u_ℓ), because F is a sum, there must exist at least one term k for which

$$f_k(x_\ell, u_\ell) > 0 \tag{34}$$

Otherwise, if all terms were zero, $\mathscr{L}(x_\ell, u_\ell)$ would be zero. Since each f_k is nonnegative and has 0 as its unique minimum, having $f_k(x_\ell, u_\ell) > 0$ implies

$$\nabla f_k(x_\ell, u_\ell) \neq 0 \tag{35}$$

In other words, the partial derivatives of f_k at (x_ℓ, u_ℓ) do not vanish. Hence the gradient of F at (x_ℓ, u_ℓ) is

$$\nabla \mathscr{L}(x_\ell, u_\ell) = \sum_{k=1}^{K} \nabla f_k(x_\ell, u_\ell) \neq 0, \tag{36}$$

since at least one ∇f_k is nonzero. This contradicts the necessary condition for a local minimum, which would require

$$\nabla \mathscr{L}(x_\ell, u_\ell) = 0. \tag{37}$$

Therefore, F cannot have a local minimum at (x_ℓ, u_ℓ) unless all terms $f_k(x_\ell, u_\ell)$ are zero.

A.4. Using our loss-function for MILPs

First, we solve the LP relaxation of the MILP to obtain a candidate solution x^*_{LP}. Suppose its objective value is

$$c^\top x^*_{LP} = O_{LP} \tag{38}$$

Then we know that the final MILP solution x^* will satisfy

$$c^\top x^* = O > O_{LP} \tag{39}$$

Next, we look for a feasible binary point x close to x^*_{LP} within the feasible region $Ax - b \leq 0$, and we impose the bounds

$$O_{LP} \leq c^\top x \leq O_{LP} \cdot \delta,$$

where $\delta > 1$. $O_{LP} = O_d$ the dual or lower bound, and $O_{LP} \cdot \delta = O_p$ the primary or upper bound.

$$\begin{aligned} &(1) \quad Ax - b \leq 0 \\ &(2) \quad O_d \leq c^\top x \leq O_p \\ &(3) \quad x_i(1 - x_i) = 0, \quad \forall i. \end{aligned} \tag{40}$$

A loss function that penalizes violations of these constraints is

$$\mathscr{L}_{\text{MILP}}(x) = \sum_{j=1}^{m} \text{ReLU}\big((Ax-b)_j\big) + \text{ReLU}\big(-c^\top x + O_d\big) + \text{ReLU}\big(c^\top x - O_p\big) + \sum_{i=1}^{n} x_i^2 \big(1 - x_i\big)^2 \tag{41}$$

The function $\mathscr{L}_{\text{MILP}}(x)$ again has no local minima, all terms are nonnegative and vanish only if x satisfies the constraints exactly (making $\mathscr{L}_{\text{MILP}}(x) = 0$).

Now we use $\mathscr{L}_{\text{MILP}}(x)$ as the loss-function for a second neural network with $O_d = O_{LP}$ and $O_p = O_{LP} \cdot \delta$. Suppose this yields a first feasible solution x_1 We now have an update of the upper bound:

$$O_p^1 = c^\top x_1 \tag{42}$$

We then use the NN a second time with $O_d = O_{LP}$ and $O_p = O_p^1$. This either returns x_1, then x_1 is optimal, or we get x_2 with

$$c^\top x_2 = O_p^2 < O_p^1 \tag{43}$$

This is then repeated until x no longer changes or until

$$\frac{x_{n-1} - x_n}{x_{n-1}} \leq \epsilon \tag{44}$$

BIBLIOGRAPHY

Shreya Arvind, Rishabh Pomaje, and Rajshekhar V Bhat. Karush-kuhn-tucker condition-trained neural networks (kkt nets). *arXiv preprint arXiv:2410.15973*, 2024. 2

Ziang Chen, Jialin Liu, Xinshang Wang, Jianfeng Lu, and Wotao Yin. On representing linear programs by graph neural networks. *arXiv preprint arXiv:2209.12288*, 2022. 5

Jian-Ya Ding, Chao Zhang, Lei Shen, Shengyin Li, Bing Wang, Yinghui Xu, and Le Song. Accelerating primal solution findings for mixed integer programs based on solution prediction. In *Proceedings of the aaai conference on artificial intelligence*, volume 34, pages 1452–1459, 2020. 6

Priya L Donti, David Rolnick, and J Zico Kolter. Dc3: A learning method for optimization with hard constraints. *arXiv preprint arXiv:2104.12225*, 2021. 7

Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018. 1, 16

Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3):121–136, 1975. 8

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019. 6

Elias B Khalil, Christopher Morris, and Andrea Lodi. Mip-gnn: A data-driven framework for guiding combinatorial solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10219–10227, 2022. 6

James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. *arXiv preprint arXiv:2103.16378*, 2021. 1

Harold W. Kuhn and Albert W. Tucker. Nonlinear programming. In Jerzy Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950*, pages 481–492. University of California Press, Berkeley and Los Angeles, 1951. 8, 19

Tae-Hoon Lee and Min-Soo Kim. Rl-milp solver: A reinforcement learning approach for solving mixed-integer linear programs with graph neural networks. *arXiv preprint arXiv:2411.19517*, 2024. 1, 2, 4, 7

Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. *arXiv preprint arXiv:2410.08288*, 2024. 8, 16

Jiacheng Lin, Jialin Zhu, Huangang Wang, and Tao Zhang. Learning to branch with tree-aware branching transformers. *Knowledge-Based Systems*, 252:109455, 2022. 7

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992. 11

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid Von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020. 1, 4, 6, 8

Chendi Qian and Christopher Morris. Towards graph neural networks for provably solving convex optimization problems. *arXiv preprint arXiv:2502.02446*, 2025. 1

Chendi Qian, Didier Chételat, and Christopher Morris. Exploring the power of graph neural networks in solving linear optimization problems. In *International Conference on Artificial Intelligence and Statistics*, pages 1432–1440. PMLR, 2024. 5, 6, 8

Lara Scavuzzo, Karen Aardal, Andrea Lodi, and Neil Yorke-Smith. Machine learning augmented branch and bound for mixed integer linear programming. *Mathematical Programming*, pages 1–44, 2024. 5

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 11

Bo Tang, Elias B Khalil, and Ján Drgoňa. Learning to optimize for mixed-integer non-linear programming. *arXiv preprint arXiv:2410.11061*, 2024. 1, 7

Haoyu Wang, Jialin Liu, Xiaohan Chen, Xinshang Wang, Pan Li, and Wotao Yin. Dig-milp: a deep instance generator for mixed-integer linear programming with feasibility guarantee. *arXiv preprint arXiv:2310.13261*, 2023. 8

Dawen Wu and Abdel Lisser. A deep learning approach for solving linear programming problems. *Neurocomputing*, 520:15–24, 2023. 1, 5, 6, 16

Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *Proceedings of the aaai conference on artificial intelligence*, volume 35, pages 3931–3939, 2021. 6, 7

Yahong Zhang, Chenchen Fan, Donghui Chen, Congrui Li, Wenli Ouyang, Mingda Zhu, and Junchi Yan. Milp-fbgen: Lp/milp instance generation with feasibility/boundedness. In *Forty-first International Conference on Machine Learning*, 2024. 8