Speeding up Logic-Based Benders Decomposition by Strengthening Cuts with Graph Neural Networks *

Johannes Varga[1 [0000−0003−1413−7115]], Emil Karlsson[2,3 [0000−0002−9498−1924]], Günther R. Raidl[1 [0000−0002−3293−177X]], Elina Rönnberg[2 [0000−0002−2081−2888]], Fredrik Lindsten[4 [0000−0003−3749−5820]], and Tobias Rodemann[5 [0000−0001−6256−0060]]

[1] Institute of Logic and Computation, TU Wien, Vienna, Austria
{jvarga,raidl}@ac.tuwien.ac.at
[2] Department of Mathematics, Linköping University, Linköping, Sweden
elina.ronnberg@liu.se
[3] Saab AB, SE-581 88 Linköping, Sweden emil.karlsson1@saabgroup.com
[4] Department of Computer and Information Science, Linköping University, Linköping, Sweden fredrik.lindsten@liu.se
[5] Honda Research Institute Europe, Offenbach, Germany
tobias.rodemann@honda-ri.de

Abstract. Logic-based Benders decomposition is a technique to solve optimization problems to optimality. It works by splitting the problem into a master problem, which neglects some aspects of the problem, and a subproblem, which is used to iteratively produce cuts for the master problem to account for those aspects. It is critical for the computational performance that these cuts are strengthened, but the strengthening of cuts comes at the cost of solving additional subproblems. In this work we apply a graph neural network in an autoregressive fashion to approximate the compilation of an irreducible cut, which then only requires few postprocessing steps to ensure its validity. We test the approach on a job scheduling problem with a single machine and multiple time windows per job and compare to approaches from the literature. Results show that our approach is capable of considerably reducing the number of subproblems that need to be solved and hence the total computational effort.

Keywords: Logic-based Benders Decomposition · Cut Strengthening · Graph Neural Networks · Job Scheduling

1 Introduction

Many different ways have been investigated lately to improve traditional combinatorial optimization methods by means of modern machine learning (ML) techniques. For a general survey see [2]. Besides more traditional approaches

* J. Varga acknowledges the financial support from Honda Research Institute Europe.

where ML techniques are used to tune parameters or configurations of an optimization algorithm or to extract problem features then utilized by an optimization algorithm [5,6], diverse approaches have been proposed where handcrafted heuristics for guiding a search framework are replaced by learned models. The advantages of successful approaches are apparent: Significant human effort may be saved by learning components of an optimization algorithm in an automated way, and these may potentially also make better decisions, leading to better performing solvers and ultimately better or faster obtained solutions. Given historical data from a practical application scenario, a more specialized model may also be trained to more effectively solve similar future instances of this problem. Moreover, if characteristics of the problem change, a learned model may also be rather easily retrained to accommodate the changes, while handcrafted methods require manual effort.

In the context of mathematical programming techniques, in particular mixed integer linear programming, learned models are used, for example, as primal heuristics for diving, to approximate the well-working but computational expensive strong branching [1], or to select valid inequalities to add as cuts by means of a faster ML model that approximates a computationally expensive look-ahead approach [14]. We focus here in particular on logic-based Benders decomposition (LBBD) [7], which is a well-known mathematical programming decomposition technique. The basic idea is that an original problem is split into a master problem and a subproblem that are iteratively solved in an alternating manner to obtain a proven optimal solution to the original problem. The master problem considers only subsets of the decision variables and constraints of the original problem and is used to calculate a solution with respect to this subset of variables. The subproblem is obtained from the original problem by using the master problem solution to fix the values of the corresponding variables. If the original objective function includes master problem variables only, the purpose of the subproblem is to augment the solution in a feasible way. If this is not possible, a so-called feasibility cut, i.e., a valid inequality that is violated by the current master problem solution, is derived and added to the master problem. Used as is, this cut would only remove a small number of master problem solutions from the master problem's solution space and as such it is not likely to yield much progress. It is therefore crucial for the practical performance of LBBD schemes to *strengthen* cuts before adding them to the master problem. Typically, such strengthening involves to solve additional subproblems.

As a first step towards speeding up the LBBD scheme through cut strengthening, we propose a learning-based approach that reduces the number of subproblems to be solved. For the learning part, subproblems are represented by graphs, and consequently we use a graph neural network (GNN) as ML model. An autoregressive approach inspired by [11] is applied to compile a promising inequality by means of iteratively selecting and adding variables under the guidance of the GNN. This GNN is trained offline on a set of representative problem instances. As test bed we use a single-machine scheduling problem with time-windows from [8]. Results clearly indicate the benefits of the GNN-guided cut

strengthening over classical strengthening strategies from the literature. In particular the number of solved subproblem instances is significantly lower, which leads to better overall runtimes to obtain proven optimal solutions.

Compared to classical Benders decomposition, which is used to solve mixed integer linear programs decomposed so that the subproblem has continuous variables, LBBD is a generalization that allows also discrete variables in the subproblem. This generalization is however done at the cost of no longer having a general strategy for deriving strong cuts from subproblem. Several different cut-strengthening techniques have been proposed in the literature, and typically they are described for the specific problem at hand. In Karlsson and Rönnberg [9], techniques for strengthening of feasibility cuts have been described within a common framework and compared on a variety of different scheduling problems to provide an overview of this area. An extension of this work to also include optimality cuts is found in the technical report [16].

It is common to apply LBBD to problems with both allocation and scheduling decisions. For example, LBBD has been applied to a single facility scheduling problem with segmented timeline [4] and an electric vehicle charging and assignment scheduling problem [17]. Other examples are [10,15].

The outline of this paper is as follows. Section 2 specifies the scheduling problem and our LBBD scheme. Section 3 describes what we aim to approximate with a GNN, how we make use of its predictions to strengthen cuts, and how we collect training data, while Section 4 discusses the specific structure of the GNN, the features used as its input, and the training procedure. Finally, Section 5 presents experimental results, and Section 6 concludes the paper.

2 Single-Machine Scheduling Problem and LBBD

In the single-machine scheduling problem we will use as test bed for our LBBD approach we are given a set of tasks I. Each task $i \in I$ may be scheduled in a non-preemptive way within one of its possibly multiple time windows $w \in W_i$ given by a release time r_{iw} and a deadline $d_{iw} \geq r_{iw}$. Task i has a processing time p_i, for which it exclusively requires the single available machine. Moreover, if task i runs before another task $j \in I$, the time difference between the start of task j and the end of task i has to be at least a given setup time s_{ij}. For each task $i \in I$ that is scheduled, a prize q_i is collected and the objective is to maximize these collected prizes.

An overview of the LBBD framework is shown in Algorithm 1. To apply LBBD to our scheduling problem, we split it into a master problem (MP), which determines the subset of tasks to be performed together with selected time windows, and a subproblem (SP) for determining an actual schedule. More specifically, the SP gets a set of task-time window pairs (TTWPs) $X \subseteq \{(i, w) : i \in I, w \in W_i\}$ as input and checks, whether the tasks can be scheduled without overlap within the selected time windows, also obeying the setup times. If such a schedule exists, it is an optimal solution to the original problem. Otherwise a feasibility cut, i.e., inequality that excludes this set of TTWPs from the MP's

Algorithm 1: LBBD scheme with cut strengthening

Data: A problem instance
Result: An optimal solution or a proof of infeasibility

```
1  F = ∅  /* Set of feasibility cuts */;
2  k ← 1  /* The current LBBD iteration */;
3  X ← ∅  /* The current TTWPs */;
4  while true do
5      X^k ← solve MP with feasibility cuts F;
6      if X^k is a feasible solution then
7          X ← X^k;
8      else
9          return problem has no feasible solution;
10     end
11     Y^k ← solve SP for TTWPs X;
12     if Y^k is a feasible solution then
13         return optimal solution (X^k, Y^k);
14     else
15         X̂ ← CutStrengthening(X);
16         F ← F ∪ X̂;
17     end
18     k = k + 1;
19 end
```

space of feasible solutions, is derived and added to the MP to prevent the same configuration from further consideration. Thus, in our case a feasibility cut is represented by a set of TTWPs.

2.1 Master Problem

We state the MP by the following binary linear program with variables $x_{iw} \in \{0,1\}$ indicating with value one that task $i \in I$ is to be performed in its time window $w \in W_i$.

$$[\mathrm{MP}(I,\mathcal{F})] \quad \max \sum_{i\in I} \sum_{q\in W_i} q_i x_{iw} \tag{1}$$

$$\text{s.t.} \sum_{w\in W_i} x_{iw} \leq 1, \qquad i \in I \tag{2}$$

$$\sum_{(i,w)\in X} (1 - x_{iw}) \geq 1, \qquad X \in \mathcal{F} \tag{3}$$

$$\sum_{i\in I} \sum_{w\in W_i(t_1,t_2)} p_i x_{iw} \leq t_2 - t_1, \qquad (t_1,t_2) \in T \tag{4}$$

$$x_{iw} \in \{0,1\}, \qquad i \in I,\ w \in W_i. \tag{5}$$

The objective (1) is to maximize the total prize collected by assigning tasks to time-windows. Constraints (2) ensure that each task is scheduled in at most

one of its time windows. All so far added feasibility cuts are represented by inequality (3), where the elements in $\mathcal{F}$ are sets of TTWPs that must not appear together in a feasible solution. Inequalities (4) are used to strengthen the MP. Set $T = \{(r_{iw}, d_{i'w'}) \mid (w, w') \in W_i \times W_{i'},\ (i, i') \in I \times I,\ d_{i'w'} > r_{iw}\}$ contains release time and deadline pairs for which a segment relaxation is used and $W_i(t_1, t_2) = \{w \in W_i \mid t_1 \le r_{iw}, d_{iw} \le t_2\}$ is the set of time windows for task $i \in I$ that starts after t_1 and ends before t_2. The inequality ensures that the total processing time of all tasks within a time interval $[t_1, t_2)$ does not exceed the interval's duration.

2.2 Subproblem

To simplify the notation, let $I(X) = \{i : (i, w) \in X\}$ denote the set of tasks to be scheduled. In the SP, the decision variables $y_i \ge 0$ represent the start time of task $i \in I(X)$. We state the SP as the following decision problem.

$$
\begin{aligned}
[\mathrm{SP}(X)] \quad & \text{find } y_i, \forall i \in I && (6)\\
\text{s.t.}\ & \textsc{Disjunctive}((y_i \mid i \in I(X)), (p_i \mid i \in I(X)),\\
& \qquad (s_{ij} \mid i, j \in I(X)), i \neq j) && (7)\\
& r_{iw} \le y_i \le d_{iw} - p_i \qquad (i, w) \in X && (8)\\
& y_i \ge 0 \qquad i \in I && (9)
\end{aligned}
$$

Constraints (7) ensure that no two tasks overlap and setup times are obeyed. Inequalities (8) guarantee that all tasks are scheduled within their time windows.

2.3 Cut Strengthening

Strengthening of feasibility cuts obtained from an infeasible subproblem is crucial for good performance of the LBBD framework. In our case, a cut is represented by a set of TTWPs X and is *strengthened* when one or more TTWPs are removed from X and the newly obtained inequality is still a valid cut in the sense that the respective SP remains infeasible. The infeasibility of the SP implies that no feasible solutions are cut away when adding the inequality to the MP. Ideally, a cut-strengthening procedure should provide one or multiple *irreducible cuts*, which are cuts that cannot be strengthened further by removing TTWPs. Should the MP become infeasible at some point, the problem instance is proven infeasible.

The deletion filter cut-strengthening algorithm is based on the deletion filter for finding an irreducible inconsistent set of constraints [3] and is used to form an irreducible cut $\hat{X}$ from a given X, see Algorithm 2. The method starts with $\hat{X} = X$ and proceeds through all TTWPs $(i, w) \in X$ one-by-one, checking the feasibility of the subproblem with (i, w) excluded. If a subproblem is infeasible, TTWP (i, w) will be permanently removed from $\hat{X}$. For example, this form of cut strengthening was already used in [15,12]. MARCO [13] is another procedure to strengthen cuts. While it is more expensive in terms of subproblems to check, it systematically enumerates *all* irreducible feasibility cuts that can be derived from X. We will use MARCO to collect training data for learning, while our cut strengthening approach is based on the faster deletion filter.

Algorithm 2: Deletion filter cut strengthening

Data: A set of TTWPs X representing a feasibility cut
Result: A set of TTWPs $\hat{X}$ representing an irreducible feasibility cut

```
1 X̂ ← X;
2 for (i, w) ∈ X do
3 |  X' ← X̂ \ {(i, w)};
4 |  if SP(X') is infeasible then
5 |  |  X̂ ← X';
6 |  end
7 end
8 return X̂;
```

3 Learning to Strengthen Cuts

We now introduce our main contribution, the graph neural network (GNN) based cut strengthening approach to speed up the LBBD scheme. The idea is to train a GNN offline on representative problem instances and apply it when solving new instances as guidance in the construction of a promising initial subset of TTWPs $\hat{X} \subseteq X$ that ideally already forms a strong irreducible cut without the need of any subproblem solving. Still, our approach then has to check if the respective $\mathrm{SP}(\hat{X})$ is feasible or not. In case the SP is feasible, further TTWPs are added one-by-one until the SP becomes infeasible. In any case, the deletion filter is finally also applied to obtain a guaranteed irreducible cut. The hope is that by this approach, only few subproblems need to be solved to "repair" the initially constructed TTWP subset in order to come up with a proven irreducible cut.

Algorithm 3 shows our GNN-based cut strengthening in more detail, which again receives a TTWP set X representing an infeasible SP and thus an initial feasibility cut as input. In its first part up to line 5, the algorithm selects a sequence of TTWPs S in an autoregressive manner steered by the GNN-based function $f_\Theta(I, X, S)$ with trainable parameters Θ. Here, I refers to the set of all tasks of the original problem, which are provided as global information, and S is the initially empty TTWP sequence. Function f_Θ returns a TTWP (i, w) that is appended to S as well as a Boolean indicator τ for terminating the construction. The way how function f_Θ is realized will be explained in Section 3.1 and how it is trained in Section 3.2. In the following while-loop, the algorithm adds further TTWPs that are again selected by function f_Θ as long as the corresponding SP remains feasible, which needs to be checked here by trying to solve the SP. Finally, in line 10, the deletion filter is applied, where the TTWPs are considered in reverse order as they have been previously selected by f_Θ.

3.1 Function $f_\Theta(I, X, S)$

Our trainable function f_Θ shall predict a best suited TTWP from X for appending it to the given TTWP sequence S as well as the termination indicator

Algorithm 3: GNN-based cut strengthening

Data: A set of TTWPs X representing a feasibility cut
Result: A set of TTWPs $\hat{X}$ representing an irreducible feasibility cut
1 $S \leftarrow []$ /* sequence of selected TTWPs */;
2 **repeat**
3 | $(i, w), \tau \leftarrow f_\Theta(I, X, S)$;
4 | append (i, w) to S;
5 **until** τ;
6 **while** *SP*(S) *is feasible* **do**
7 | $(i, w), \tau \leftarrow f_\Theta(I, X, S)$;
8 | append (i, w) to S;
9 **end**
10 $\hat{X} \leftarrow$ Deletion Filter(reverse(S));
11 **return** $\hat{X}$;

τ, so that in the ideal case the final S represents an irreducible feasibility cut that is also *strong* in the sense that it is likely binding in the final MP iteration. In a first step of $f_\Theta(I, X, S)$, a complete directed graph $G = (X, A)$ with nodes corresponding to the TTWPs in X is set up, and node as well as arc features are derived from I, X, and S, cf. Section 4. Following in parts the autoregressive approach in [11], we then apply a GNN to graph G, which returns for each node and thus each TTWP $(i, w) \in X \setminus S$ a value $p_{iw} \in [0, 1]$ that shall approximate the probability that, under the assumption that S does not contain a feasibility cut yet, when appending (i, w) to S, this set is getting one step closer to containing some strong cut while including as few TTWPs not being part of this cut as possible. More precisely, let $\mathcal{X}$ be the set of all irreducible strong cuts being subsets of X. Moreover, let $\delta(S, \mathcal{X}) = \min_{\hat{X} \in \mathcal{X}} |S \setminus \hat{X}|$ be the minimum number of TTWPs any complete extension of S towards a feasibility cut must contain that will not be part of a strong cut. Subset $\mathcal{X}' = \{\hat{X} \in \mathcal{X} : |S \setminus \hat{X}| = \delta(S, \mathcal{X})\}$ then contains those strong cuts that might be created by extending S without increasing $\delta(S, \mathcal{X})$. Values p_{iw} should therefore approximate

$$\hat{p}_{iw} = \begin{cases} 1 & \text{if } (i, w) \in \bigcup_{\hat{X} \in \mathcal{X}'} \hat{X} \\ 0 & \text{else.} \end{cases} \tag{10}$$

We intentionally leave p_{iw} undefined for the case that S already contains an irreducible cut, although in practice f_Θ might indeed be called for such cases if the termination prediction is not precise enough. These predicted values of p_{iw} are, however, not critical as the corresponding unnecessarily appended TTWPs in S will get removed by the finally applied deletion filter anyway.

In addition to the above p_{iw} value, the GNN further returns a value $\tau_{iw} \in [0, 1]$ for each TTWP $(i, w) \in X \setminus S$ approximating the probability that when appending (i, w) to S, this augmented set contains already an irreducible feasibility cut. Function $f_\Theta(I, X, S)$ evaluates the described GNN, selects a TTWP $(i, w) \in X \setminus S$ with maximum value p_{iw}, and returns (i, w) together with

$\lceil \tau_{iw} + 0.5 \rfloor$ as Boolean valued τ. Ties among p_{iw} are broken in favor of higher values τ_{iw} and finally at random.

3.2 Collecting Training Data

The GNN is trained in an offline manner by supervised learning on a substantial set of representative problem instances. To obtain labeled training data, the LBBD framework is applied to each of these problem instances with the following extensions in order to identify strong irreducible feasibility cuts. In the first step, the LBBD scheme is performed with MARCO as cut-strengthening approach to enumerate all possible irreducible cuts for each infeasible subproblem. Let us denote with $\mathcal{F}$ the set of cuts obtained in this way over a full LBBD run. The second step finds an irreducible subset of *strong* cuts of the form $\mathcal{X} \subseteq \mathcal{F}$ that is actually required to get a feasible optimal solution in the master problem in the sense that the master problem will give an infeasible solution when removing one of those cuts. This is done by applying a deletion filter on $\mathcal{F}$ using a random order, iteratively removing cuts and checking whether the master problem still gives a feasible solution. Note that there might be other minimal sets of irreducible cuts, which we do not encounter with this approach.

For each training instance I we perform the above procedure and collect for each Benders iteration a tuple $(I, X, \mathcal{F}, \mathcal{X})$ where X denotes the TTWPs of the SP and thus original feasibility cut and $\mathcal{X} = \{\hat{X} \in \mathcal{F} : \hat{X} \subseteq X \wedge \hat{X} \text{ is strong}\}$ is the set of strong cuts that can be derived from X. These tuples are then used to derive training samples for the GNN.

This is done by performing for each tuple $(I, X, \mathcal{F}, \mathcal{X})$ rollouts that simulate the derivation of feasibility cuts without needing to actually solve any further subproblems. Algorithm 4 details this rollout procedure. In each iteration of the loop, step 7 selects a TTWP that brings S one step closer to containing a strong cut from $\mathcal{X}$ and adds it to S. This is repeated until S is a superset of an irreducible cut. Note that in this procedure, the order of the elements in S does not matter, and we therefore realize it as a classical set in contrast to Algorithm 3, where S needs to be an ordered sequence. Additionally, in each iteration the training, labels $\hat{p} = (\hat{p}_{iw})_{iw \in X}$ and $\hat{\tau} = (\tau_{iw})_{iw \in X}$ are calculated based on the definitions of p_{iw} and τ_{iw} from Section 3.1 and a training sample is added. The notation $[\![\cdot]\!]$ denotes Iverson brackets, i.e., $[\![C]\!]$ is 1 if the condition C is true and 0 otherwise.

4 Graph Neural Network

The architecture of our GNN is shown in Figure 1. Remember that the nodes of our graph correspond to the TTWPs of an original feasibility cut X. For each node $(i, w) \in X$, vector x_{iw} shall be the vector containing all node features and x the respective matrix over all nodes. An initial node embedding h_0 of size 64 per node is derived by a linear projection with trainable weights. Moreover, for

Algorithm 4: The rollout procedure to construct training samples from a tuple $(I, X, \mathcal{F}, \mathcal{X})$.

Data: Original feasibility cut X, all irreducible cuts $\mathcal{F}$, strong cuts $\mathcal{X}$.
Result: A set T of training samples, a sample being a tuple $(I, X, S, \hat{p}, \hat{f})$, where I is the set of all tasks with their attributes and $\hat{p}$ and $\hat{f}$ are the labels.

1 $S \leftarrow \emptyset$;
2 $T \leftarrow \emptyset$;
3 **repeat**
4 $\quad \hat{p}_{iw} \leftarrow [\![(i,w) \in \bigcup_{\hat{X} \in \mathcal{X}'} \hat{X}]\!] \qquad \forall (i,w) \in X \setminus S$;
5 $\quad \hat{f}_{iw} \leftarrow [\![\exists \hat{X} \in \mathcal{F} : \hat{X} \subseteq S \cup \{(i,w)\}]\!] \qquad \forall (i,w) \in X \setminus S$;
6 $\quad T \leftarrow T \cup \{(I, X, S, \hat{p}, \hat{f})\}$;
7 $\quad$ select $(i,w) \in \bigcup_{\hat{X} \in \mathcal{X}'} \hat{X}$ uniformly at random;
8 $\quad S \leftarrow S \cup \{(i,w)\}$;
9 **until** $\exists \hat{X} \in \mathcal{F} : \hat{X} \subseteq S$;
10 **return** T;

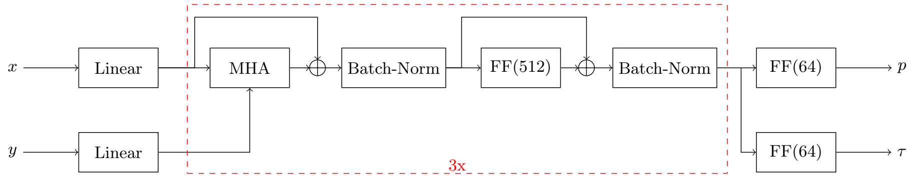

Fig. 1: Architecture of the GNN.

all arcs in A and their features, we also derive an arc embedding h^A of size 32 per arc by a trainable linear projection.

We apply a transformer-based convolution layer that also considers arc embeddings, implemented in Pytorch Geometric, with 8 heads, skip connections, and batch normalization, followed by a feed-forward layer with a hidden sublayer of dimensionality 512, parametric rectified linear units (PReLUs) as nonlinearity, and again with skip connections and batch normalization, similar as in the encoder part of [11] and usually done in transformers [18]. This multihead attention layer plus feed-forward layer is repeated $k = 3$ times. We chose hyper-parameters by hand based on preliminary experiments and guided by [11].

In contrast to transformers and [11], we do not follow an encoder/decoder architecture, but keep things simpler. To obtain output values p_{iw} and τ_{iw} for each node (i,w), we further process the so far obtained node embeddings h^k_{iw} by two independent feed-forward neural networks with one hidden layer of dimensionality 64 and single final output nodes with sigmoid activation functions, respectively. In the autoregressive approach, the whole GNN is completely evaluated in each step with updated features, in particular concerning the already selected TTWPs. Output nodes for already selected TTWPs are masked out.

As we do binary classification, we use binary cross-entropy as loss function:

$$\mathcal{L}(p, \hat{p}, \tau, \hat{\tau}) = \frac{1}{2}(\mathcal{L}(p, \hat{p}) + \mathcal{L}(\tau, \hat{\tau})) \tag{11}$$

with

$$\mathcal{L}(p, \hat{p}) = \frac{1}{|V \setminus S|} \sum_{v \in V \setminus S} -(\hat{p}_v \log p_v + (1 - \hat{p}_v) \log(1 - p_v)) \tag{12}$$

and

$$\mathcal{L}(\tau, \hat{\tau}) = \frac{1}{|V \setminus S|} \sum_{v \in V \setminus S} -(\hat{\tau}_v \log \tau_v + (1 - \hat{\tau}_v) \log(1 - \tau_v)). \tag{13}$$

The list of node and arc features we use is given in Table 1. There are three additional features with the values from the last three arc features from the reverse arc. Features corresponding to points in time are scaled with function

$$\text{time-scaling}(t) = (t - \min_{i,w}(r_{iw}))/(\max_{i,w}(d_{iw}) - \min_{i,w}(r_{iw})), \tag{14}$$

over all tasks i and their time windows w in the whole set of training instances. All other features are either normalized across the training data to values within [0,1] or kept if they are already reasonably distributed. We further compute a basic schedule from the TTWPs in X with a variation of the earliest deadline first (EDF) rule. This greedy heuristic selects the task with the earliest deadline of only those tasks, whose release time is before the ending time of the previously scheduled task. If all unscheduled tasks have a later release time, the unscheduled task with the next release time is scheduled next. Some of the features are based on the start times s_i^{EDF} and end times e_i^{EDF} of the tasks $i \in I$ in this schedule.

5 Experimental Evaluation

We implemented our approach in Python 3.10.8 using `pytorch_geometric`[6] for the GNN and used the MILP solver Gurobi[7] 9.5 and the CP solver CPOptimizer[8] 20.1 to solve the master and the subproblem instances, respectively. To evaluate our approach we use benchmark instances that are generated as described in [8] for the avionics application with $m = 3$ resources. Note that the problem formulation in [8] differs in having secondary resources, but we are able to model this aspect with setup times. We consider instance sizes $n \in \{10, 15, 20, 25\}$, for which we are able to find optimal solutions by the LBBD approach in most cases in reasonable time.

For each instance size between 300 000 and 600 000 training samples were collected from 2 000 to 200 000 independent instances, see Table 2 for details on the training. In each of 40 to 300 epochs we train with minibatches of 64 samples,

[6] https://pytorch-geometric.readthedocs.io/
[7] https://www.gurobi.com/
[8] https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer

Table 1: Node and arc features. For a node feature (i, w) denotes the TTWP associated with the node, for an arc feature (i_1, w_1) represents the TTWP of the source node and (i_2, w_2) the TTWP of the target node.

Formula	Type	Normalization	Meaning
$[\![(i,w) \in S]\!]$	node	-	Whether the TTWP has already been selected
$\lvert S\rvert / \lvert X\rvert$	node	-	Fraction of selected TTWPs
p_i	node	normalized	Processing time
r_{iw}	node	time-scaling	Release time
d_{iw}	node	time-scaling	Deadline
$d_{iw} - r_{iw}$	node	normalized	Length of the time window
$q_i / \max_{j \in I}(q_j)$	node	-	Relative prize
$d_{iw} - r_{iw} - p_i + 1$	node	normalized	Absolute flexibility of scheduling the job
$1 - p_i/(d_{iw} - r_{iw})$	node	-	Relative flexibility of scheduling the job
$[\![d_{iw} - r_{iw} > p_i]\!]$	node	-	Whether there is any flexibility in scheduling the job
s_i^{EDF}	node	time-scaling	EDF start time
e_i^{EDF}	node	time-scaling	EDF end time
$[\![s_i^{\text{EDF}} = r_i]\!]$	node	-	Whether the EDF start time is at the release time
$[\![e_i^{\text{EDF}} = d_i]\!]$	node	-	Whether the EDF end time is at the deadline
$[\![e_i^{\text{EDF}} > d_i]\!]$	node	-	Whether the task is late in the EDF schedule
$\max(e_i^{\text{EDF}} - d_i, 0)$	node	time-scaling	Lateness of the task in the EDF schedule
$s_{i_1 i_2}$	arc	normalized	Setup time
$[\![s_{i_2}^{\text{EDF}} = e_{i_1}^{\text{EDF}} + s_{i_1 i_2}]\!]$	arc	-	Whether the two tasks cannot be any closer than in the EDF schedule
$\max(s_{i_2}^{\text{EDF}} - e_{i_1}^{\text{EDF}}, 0)$	arc	normalized	Time difference of the EDF schedule
-	arc	normalized	Overlap of the time windows
$[\![r_{i_1 w_1} \leq r_{i_2 w_2} < d_{i_1 w_1}]\!]$	arc	-	Whether time window w_2 starts within time window w_1
$[\![s_{i_1 i_2} > 0]\!]$	arc	-	Whether there is a non-zero setup time
-	arc	-	Whether task i_2 is the direct successor of task i_1 in the EDF schedule

Table 2: Characteristics of the training.

n	10	15	20	25
# of training instances	200 000	60 000	20 000	2 000
# of training samples	547 893	354 030	339 263	322 955
# of epochs	40	40	200	400
# of test instances	20 000	6 000	2 000	200
# of test samples	54 723	35 178	33 493	37 832
Recall of p-values [%]	92.00	82.92	80.91	81.15
Precision of p-values [%]	91.44	86.44	80.32	80.64
Recall of τ-values [%]	98.07	94.66	86.17	74.55
Precision of τ-values [%]	93.00	90.21	86.75	68.81

using the ADAM optimizer with a learning rate of $3 \cdot 10^{-4}$. For regularization we use dropout before each transformer-based graph convolution layer, before each feed-forward layer, and on the normalized attention coefficients with a dropout probability of 25% for $n \in \{10, 15, 20\}$ and 10% for $n = 25$.

First, we compare the performance of the LBBD with our approach as cut-strengthening procedure, denoted by *GNN*, to the deletion filter where the order of the TTWPs is (a) random (*Random*), (b) dictated by the instance (*Sorted*), or (c) sorted by decreasing slack as done by Coban and Hooker [4] (*Hooker*). For this comparison we ran each approach for each benchmark instance on a single core of an AMD Ryzen 9 5900X without using a GPU. The results for $n = 20$ and $n = 25$ are shown in Figure 2 in the form of cumulative distribution plots. Each plot shows how many instances could be solved to optimality out of 100, with which number of subproblems to be solved, in which time, and within how many major LBBD iterations. As can be seen, our GNN-based cut-strengthening procedures significantly reduce the number of subproblems that have to be solved as well as the total runtimes. For example, for $n = 20$ *GNN* on average only needs to solve $\approx 30.4\%$ of the number of subproblems *Hooker* has to solve and requires only $\approx 54\%$ of *Hooker*'s time. The number of LBBD iterations is similar for all methods, which means that the time savings are only achieved by reducing the number of subproblems that had to be solved.

Progress of the bounds. In each iteration of the LBBD we obtain a dual bound from the solution to the master problem. Concerning primal bounds, note that any set of TTWPs for which the subproblem is feasible represents a feasible solution to the overall problem. We use as primal bound the value of the best such solution encountered during the cut strengthening procedure. The average development of those primal and dual bounds for $n = 20$ and $n = 25$ tasks are shown in Figure 3. Observe that the GNN-based approach performs significantly better in respect to the dual bound with an average percentage gap to the optimal solution of 4.85% after four seconds for $n = 25$ compared to 10.6% for *Hooker*. However the primal bounds perform mostly worse. This can be explained by the fact that the subproblem for the GNN-based cut strengthening is only called

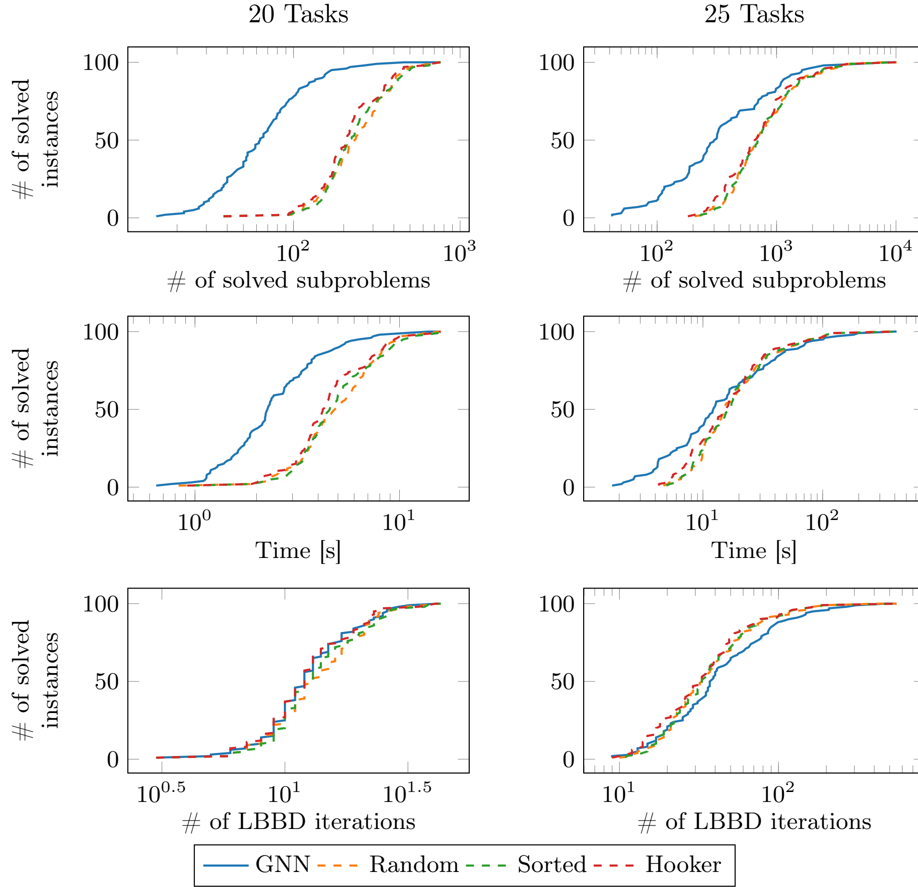

Fig. 2: Cumulative distribution diagrams with the number of solved instances over the number of subproblems solved, the running time and the number of Benders iterations performed.

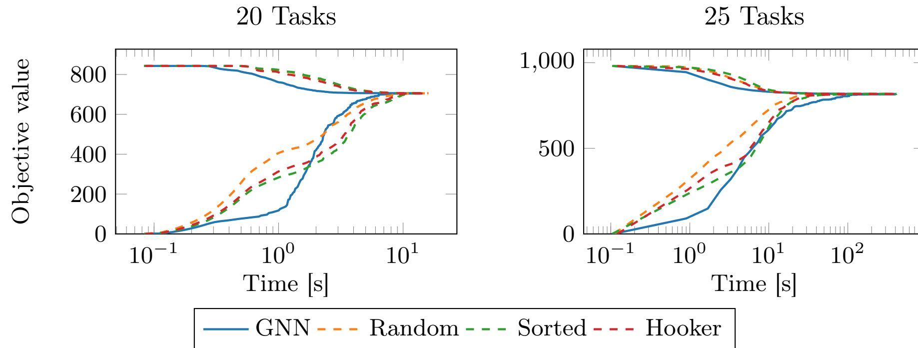

Fig. 3: Primal and dual bounds of the approaches over time.

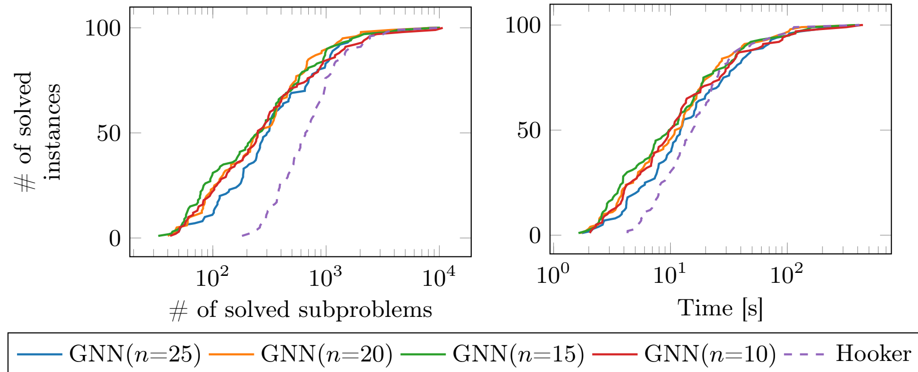

Fig. 4: Cumulative distribution diagrams showing the performance of the models trained on instances of size $n = 10$, $n = 15$, $n = 20$ and $n = 25$, respectively, on instances of size $n = 25$ and comparing to the performance of *Hooker*.

for smaller sets of TTWPs, which have a worse objective value. We remark that we do not focus here on getting good heuristic solutions and thus primal bounds early. To improve on this aspect, it would be natural to include some primal heuristics, such as a local search or more advanced metaheuristic for strengthening intermediate heuristic solutions.

Out-of-distribution generalization. So far we applied *GNN* only to instances of the same size as the neural network model has been trained with. Now, we investigate the out-of-distribution generalization capabilities in the sense that the models trained on instances with $n \in \{10, 15, 20\}$ tasks, respectively, are applied to larger instances with 25 tasks. Figure 4 shows the corresponding cumulative distribution plots for *GNN* and compares them to *Hooker*. The generalization works surprisingly well and *GNN* trained with $n = 15$ works even slightly better than the model trained with $n = 25$, requiring a geometric mean time that is only $\approx 76\%$ of *Hooker*'s.

6 Conclusion

We trained GNNs to guide the cut-strengthening in logic-based Benders decomposition. An autoregressive approach is used, where the GNN first constructs a preliminary inequality, which is postprocessed to ensure that it is indeed an irreducible cut. The approach is tested on a single machine scheduling problem with time windows. For this problem the number of Benders subproblems solved can be reduced down to one third and up to half of the runtime can be saved on average. It is up to future work to scale up the training of the GNN to larger instance sizes. Results suggest that generalization of the GNN to larger instance sizes works quite well and thus curriculum learning seems to be promising.

References

1. Alvarez, A.M., Louveaux, Q., Wehenkel, L.: A machine learning-based approximation of strong branching. INFORMS Journal on Computing **29**(1), 185–195 (2017)
2. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: A methodological tour d'horizon. European Journal of Operational Research **290**(2), 405–421 (2021)
3. Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. ORSA Journal on Computing **3**(2), 157–168 (1991)
4. Coban, E., Hooker, J.N.: Single-facility scheduling by logic-based Benders decomposition. Annals of Operations Research **210**, 245–272 (2013)
5. Friess, S., Tiňo, P., Xu, Z., Menzel, S., Sendhoff, B., Yao, X.: Artificial neural networks as feature extractors in continuous evolutionary optimization. In: 2021 Int. Joint Conference on Neural Networks. pp. 1–9 (2021)
6. Gräning, L., Jin, Y., Sendhoff, B.: Efficient evolutionary optimization using individual-based evolution control and neural networks: A comparative study. In: ESANN. pp. 273–278 (2005)
7. Hooker, J.N., Ottosson, G.: Logic-based Benders decomposition. Mathematical Programming **96**, 33–60 (2003)
8. Horn, M., Raidl, G.R., Rönnberg, E.: A* search for prize-collecting job sequencing with one common and multiple secondary resources. Annals of Operations Research **307**, 477–505 (2021)
9. Karlsson, E., Rönnberg, E.: Strengthening of feasibility cuts in logic-based Benders decomposition. In: Stuckey, P.J. (ed.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 45–61. Springer (2021)
10. Karlsson, E., Rönnberg, E.: Logic-based Benders decomposition with a partial assignment acceleration technique for avionics scheduling. Computers & Operations Research **146**, 105916 (2022)
11. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems! In: Int. Conference on Learning Representations (2019)
12. Lam, E., Gange, G., Stuckey, P.J., Van Hentenryck, P., Dekker, J.J.: Nutmeg: a MIP and CP hybrid solver using branch-and-check. SN Operations Research Forum **1**(3), 22 (2020)
13. Liffiton, M.H., Malik, A.: Enumerating infeasibility: Finding multiple MUSes quickly. In: Gomes, C., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. LNCS, vol. 7874, pp. 160–175. Springer (2013)
14. Paulus, M.B., Zarpellon, G., Krause, A., Charlin, L., Maddison, C.: Learning to cut by looking ahead: Cutting plane selection via imitation learning. In: Proceedings of the 39th Int. Conference on Machine Learning. pp. 17584–17600. PMLR (2022)
15. Riedler, M., Raidl, G.R.: Solving a selective dial-a-ride problem with logic-based benders decomposition. Computers & Operations Research **96**, 30–54 (2018)
16. Saken, A., Karlsson, E., Maher, S.J., Rönnberg, E.: Computational evaluation of cut-strengthening techniques in logic-based Benders' decomposition, under review
17. Varga, J., Raidl, G.R., Limmer, S.: Computational methods for scheduling the charging and assignment of an on-site shared electric vehicle fleet. IEEE Access **10**, 105786–105806 (2022)
18. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)