Learning Precedences for Scheduling Problems with Graph Neural Networks

Hélène Verhaeghe[1]
DTAI, KU Leuven, Belgium

Quentin Cappart
Polytechnique Montréal, Canada

Gilles Pesant
Polytechnique Montréal, Canada

Claude-Guy Quimper
Université Laval, Quebec, Canada

Abstract

The *resource constrained project scheduling problem* (RCPSP) consists of scheduling a finite set of resource-consuming tasks within a temporal horizon subject to resource capacities and precedence relations between pairs of tasks. It is $\mathcal{NP}$-hard and many techniques have been introduced to improve the efficiency of CP solvers to solve it. The problem is naturally represented as a directed graph, commonly referred to as the *precedence graph*, by linking pairs of tasks subject to a precedence. In this paper, we propose to leverage the ability of *graph neural networks* to extract knowledge from precedence graphs. This is carried out by learning new precedences that can be used either to add new constraints or to design a dedicated variable-selection heuristic. Experiments carried out on RCPSP instances from PSPLIB show the potential of learning to predict precedences and how they can help speed up the search for solutions by a CP solver.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Machine learning; Mathematics of computing → Combinatorial optimization

Keywords and phrases Scheduling, Precedence graph, Graph neural network

Digital Object Identifier 10.4230/LIPIcs.CP.2024.30

Funding This research received funding from IVADO and the Canada First Research Excellence Fund / Apogée fund PostDoc-2022-2378128196 and in part by NSERC Discovery Grants 218028/2017. It also received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 101070149, project Tuples.

Acknowledgements We thank the anonymous reviewers for their constructive criticism which helped us improve the original version of the paper.

1 Introduction

Scheduling problems arise in many fields, from assembling planes to scheduling maintenance tasks. Constraint programming (CP) has been successfully used to solve many types of scheduling problems [35, 29]. This is mainly due to the combination of global constraints and efficient dedicated heuristics used when solving such problems. When they are subject to precedence constraints, scheduling problems are often $\mathcal{NP}$-hard [20]. Nevertheless, every such precedence may help to improve the inferences made by global constraints. Precedences are naturally represented as a directed graph by linking two tasks subject to a precedence. *Graph neural networks* (GNNs) [28] are designed to learn from graph-structured data, including

[1] The first author was affiliated to Polytechnique Montréal during the majority of this work.

deciding whether an edge is present. A natural question our paper tries to answer is then: Can GNNs help us identify additional precedences? And as a follow-up question: How useful can these learned precedences be?

Following this idea, this paper proposes to leverage the ability of GNNs to learn new precedences between tasks for the standard *resource constrained project scheduling problem* (RCPSP) [26]. These precedences are used to enhance the solving process of a CP solver, both by adding new constraints in the model and by designing a dedicated variable-selection heuristic. The learning is carried out from a precedence graph representation of the RCPSP and the GNN is trained using best-so-far solutions found by solving, up to a fixed time limit, a set of instances specially considered for training. The models are trained and evaluated on RCPSP instances from PSPLIB [19]. The results obtained show the promise of learning to predict precedences and their relevance for speeding up the search for solutions by a CP solver.

Our approach shows that it is possible to learn precedences using a simple GNN architecture. In synergy with the dynamic metaheuristic SBPS [9, 36] and VSIDS [24], the learned precedences manage to improve our baseline (i.e., solving the problem without any additional precedences). Using the learned preferences as additional constraints allows to get better bounds, but comes with the drawback of potentially deteriorating the optimum for each wrong precedence added. Using the preferences as a piece of information to drive the search preserves the optimum while leading to better first solutions.

The paper is structured as follows. The next section presents the technical background regarding existing solving processes for the RCPSP, graph neural networks, and the learning of heuristics. Then the methodology we introduce to learn and leverage precedences is described. Finally the experiments carried out and the results obtained are discussed.

2 Technical Background and Related Work

2.1 Resource Constrained Project Scheduling Problem

An instance of the *resource constrained project scheduling problem* (RCPSP) [26] consists of a finite set of n tasks (or activities) T to be executed with the help of a finite set of m resources R. Each resource $r \in R$ has a finite capacity C_r, and each task $i \in T$ has a starting time s_i, an ending time e_i, and is executed without interruption during p_i units of time (i.e. we have $s_i + p_i = e_i$) while using c_{ir} units of each resource r. In addition, there are precedences between some pairs of tasks. We say that task i precedes task j if the execution of i should be finished before starting the execution of j (i.e., we have $e_i \leq s_j$).

We call *precedence graph* of an RCPSP instance the directed graph $P = (V, E)$ where there exists an injective function $M : T \to V$ mapping each task to a vertex. If and only if task i precedes task j, there exists an arc going from $M(i)$ to $M(j)$. We call the *transitive closure* of an RCPSP instance the set of precedences which also contains all the transitive precedences: given task i, j and k, if the instance defines that i precedes j and j precedes k, we know by transitivity that i precedes k. We call "i precedes k" a transitive precedence. The *transitive closure precedence graph* is thus the precedence graph representing the transitive closure of the instance. The `cumulative` constraint is one of the main components of solving RCPSP problems using CP. Its filtering algorithms [35, 29] prevent the resources from being overloaded.

Many heuristics are efficient in solving the RCPSP problem [8]. The most effective are *variable state independent decaying sum* (VSIDS) [24] and *solution-based phase saving* (SBPS) [9, 36]. The heuristic VSIDS uses counters for each variable. Each counter is incremented when a constraint involving the variable is generated. The choice of variable and value is based on the values of these counters. On the other hand, SBPS is only a value-selection heuristic. Each time a solution is found, the values are stored and when a variable is selected, the value to branch on is the one used in the last solution found. We note that both heuristics can be used together.

2.2 Graph Neural Network

A *graph neural network* (GNN) [28] is a specific neural network architecture dedicated to computing a vector representation, also referred to as *embedding*, for each node of a graph given as input [28, 38] (e.g., a precedence graph). This is carried out by aggregating several times information from neighboring nodes. Each aggregation operation is referred to as a *layer* of the GNN and involves weights that must be learned. This operation can be performed in many ways, and there exist in the literature different variants of GNNs, such as graph attention network [34] or gated graph sequence neural network [21]. Our work is based on the well-known *GraphSAGE* architecture [15], which can efficiently generate representations for graphs unseen during the training phase. Formally, let $G = (V, E)$ be a graph where V is the set of vertices, and where E is the set of edges. A GNN is composed of K layers. Let $h_v^k \in \mathbb{R}^d$ be a d-dimensional vector representation of a node $v \in V$ at layer k, and let $h_v^{k+1} \in \mathbb{R}^l$ be a l-dimensional vector representation of v at the next layer. We highlight that the dimension of the representation can be different at each layer (d for the layer k, and l for the layer $k+1$). The inference process of a GNN consists in computing the next representation (h_v^{k+1}) from the previous one (h_v^k) for each node v. The fundamental equations of *GraphSAGE* are given in Equations (1) to (3):

$$a_v^{k+1} = \mathsf{AGGREG}\left(h_u^k \;\middle|\; u \in N(v)\right) \qquad \forall v \in V \qquad (1)$$

$$c_v^{k+1} = g\left(\Theta^k \times \mathsf{CONCAT}\left(h_v^k, a_v^{k+1}\right)\right) \qquad \forall v \in V \qquad (2)$$

$$h_v^{k+1} = \frac{c_v^{k+1}}{\|c_v^{k+1}\|_2} \qquad \forall u \in V \qquad (3)$$

Three operations are carried out. First, Eq. (1) aggregates information from neighbors for each node. This is often done by taking the mean of each value (*mean-pooling*) or their sum (*sum-pooling*). A key aspect of *GraphSAGE* aggregation is that the neighborhood function $N(v)$ gives a fixed-size random subset of nodes sampled from the whole neighborhood of node v. A new subset is randomly sampled at each layer. Second, Eq. (2) concatenates the current representation of a node $h_v^k \in \mathbb{R}^d$ with the aggregated representation of its neighbors $a_v^{k+1} \in \mathbb{R}^d$. This vector is then linearly transformed by weight matrix $\Theta^k \in \mathbb{R}^{l \times 2d}$ which is learned during the training phase through backpropagation. A non-linear transformation g such as ReLU is subsequently performed [13]. Third, the value obtained (c_v^{k+1}) is normalized using the Euclidean norm, and the node representation at layer $k+1$ is obtained (h_v^{k+1}). The process terminates after K layers, and a final representation (h_v^K) is obtained for each node v in the graph. Such representations can after that be used for different graph-related tasks, such as classification [11], link prediction [22, 39, 2], or combinatorial tasks [3]. In another context, GNNs are increasingly considered in combinatorial optimization [3], either directly as an end-to-end solver [17], or as a mechanism to enhance existing solvers [12, 4].

Task	p_i	c_{ir_1}	c_{ir_2}	succ
A	2	1	2	B C D
B	3	2	2	E
C	1	1	2	
D	2	2	1	C
E	1	1	1	C

(a)

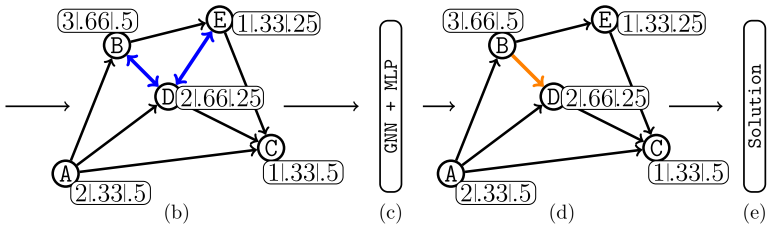


Figure 1 Overview of the methodology introduced (on a toy example with $|R| = 2$). (a) An RCPSP instance, with $C_{r_1} = 3$ and $C_{r_2} = 4$. (b) The instance is transformed into a graph with the features of each node (time, normalized usage of r_1, and r_2). The blue arcs are the candidate arcs. (c) The GNN takes the graph as input and outputs an embedding used by the MLP to provide the prediction. (d) New predicted edges for the graph. (e) Prediction is used to find a solution.

2.3 Learning Heuristics in Constraint Programming

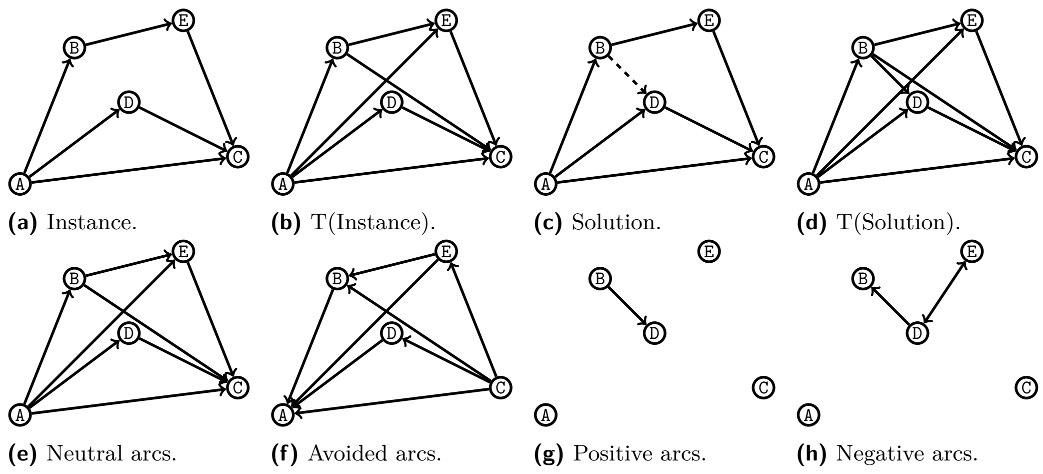
Designing branching heuristics for CP thanks to machine learning has already been considered in related works, either to learn a variable-selection or a value-selection heuristic. For instance, [30] proposed to combine reinforcement learning and graph neural networks to learn the next variable to branch for constrained satisfaction problems. In [32], graph neural networks were leveraged to initialize such a heuristic for a hybrid CP-SAT solver. Learning has also been considered for online settings by [10]. Concerning the value-selection heuristic, [4] introduced a framework able to leverage models trained with reinforcement learning inside a CP solver. It is done thanks to a recursive formalization. This idea has been further extended by [5, 23] who proposed to carry out the learning inside the solver. A general framework for learning value heuristics, and relevant features for training the model, has been also introduced by [7]. Compared to these works that aim to generate heuristics for a large range of problems and do not achieve state-of-the-art performance, our contribution is focused on scheduling problems and strive to improve upon competitive CP models. To do so, features specific in scheduling (i.e., precedence graphs) are leveraged. Recently, [31] proposed an approach close to ours. In their paper, they construct a graph with nodes representing either a task or a resource. Using GNNs, they predict the starting time of each of the tasks, then construct an ordering of the tasks based on these predicted starting times and finally use an algorithm (serial schedule generation scheme) to construct a feasible solution with respect to this ordering. They tackle classical RCPSPs and stochastic RCPSPs.

3 Enhancing CP with Learned Precedences

This section introduces the core contribution of the paper. It describes the complete methodology we designed to learn relevant precedences and how they can be used to speed up a CP solver. The methodology is based on the hypothesis that the knowledge of precedences belonging to a high-quality solution is information that is relevant to solving instances faster. Inside a CP solver, this can be achieved by enabling additional propagation (e.g., with additional constraints), or by directing the search. An overview of the methodology is illustrated in Fig. 1. The following sections describe each part of the methodology.

3.1 Step 1: Training Set Construction

The first step is to build a relevant dataset to train the model. To do so, the main idea is to: (1) solve a large number of RCPSP instances with an expensive solving process (e.g., a CP solver), (2) consider all the precedences obtained in the solutions generated, (3) use



(a) Instance. **(b)** T(Instance). **(c)** Solution. **(d)** T(Solution).
(e) Neutral arcs. **(f)** Avoided arcs. **(g)** Positive arcs. **(h)** Negative arcs.

Figure 2 Example of all the types of arcs for the same toy example as in Figure 1.

this information as data to train the model. The expected outcome is to obtain quickly precedences occurring in high-quality solutions of unseen instances, without requiring the execution of the expensive solving process. The learning is supervised as it requires a ground truth. Let $P = (V, E)$ be a precedence graph as defined previously (for example, the precedence graph defined in Fig. 2a), and let $D : \{(x^{(i)}, y^{(i)}), \dots, (x^{(n)}, y^{(n)})\}$ be the training set we need to build to train the model. In the machine learning terminology, $x^{(i)}$ corresponds to a specific *input* ($i \in \{1, \dots, n\}$), and $y^{(i)}$ corresponds to the *target value* associated to that input.

Let s^P be a solution of P, obtained by any solving process (Fig. 2c is the optimal solution in our example). We note that this solution corresponds to a more constrained precedence graph than P, with the same number of nodes, and more edges. A first important design choice is to determine *which solutions should be considered for the training?*

A natural idea is to take the optimal solution of previously solved instances. However, this is barely feasible in practice, as it assumes that there exists a solving process able to prove the optimality of these solutions in a reasonable time, which is intractable for challenging instances. For such a reason, we relaxed this idea and considered the best solution obtained instead, up to a predetermined timeout as the base for our first training set.

Then, we define $T(s^P)$ as the transitive closure of s^P (for example, Fig. 2d). This corresponds to a precedence graph $P' = (V', E')$ where $V' = V$ and $E' \supseteq E$. Before their integration in the training set, the solutions are transformed by their transitive closure (for example, Fig. 2d). This is done to reduce the diameter of the graphs. The intuition behind this design choice is that high-diameter graphs usually require deeper and more expensive GNNs to be efficient [6]. Each node V' is decorated with two types of features, recording information about the instances. They are as follows:

1. The processing time p_i of the task linked to node $i \in V'$.
2. The normalized usage of each resource r, i.e. the usage of the task i (c_{ir}) divided by the total availability of this resource within the instance (C_r).

As the instances we considered involve four resources each, we have five features for each node. Other features were considered (e.g., the proportion of the different resources used) but were not selected in the final model as they did not improve the performance. Finally, let us consider the set of pairs between vertices in P'. Each pair corresponds to a specific relation between two tasks in a solution. We identified four possible relations. Given $P = (V, E)$ (i.e., the transitive closure precedence graph of the instance) and $P' = (V', E')$ (i.e., the graph corresponding to the transitive closure precedence graph of a solution s^P, also referred to as $T(s^P)$), the four relations are as follows:

1. *Neutral arcs* (e.g., Fig. 2e): these arcs correspond to the ones included in E. Intuitively, such arcs relate to specific settings of the problem. They define the structure of the graph but are not relevant for computing the training loss.
2. *Avoided arcs* (e.g., Fig. 2f): considering all the arcs $(v, u) \in E$, the avoided arcs correspond to the set of arcs (u, v). Intuitively, we know that these arcs will never be part of a feasible solution because of the structure of the problem. Such arcs are not relevant for the loss either.
3. *Positive arcs* (e.g., Fig. 2g): these arcs correspond to the ones included in the set $E' \setminus E$. It corresponds to the additional precedences that are present in the solution obtained and missing in the initial graph. They are the arcs we want to be able to predict and should be used for the training.
4. *Negative arcs* (example Fig. 2h): these arcs correspond to all the other arcs that do not belong to any of the last three sets. Such arcs represent negative samples that can be used for training. The model should be able to predict that these arcs should not be part of a good solution.

In our first experiments, we considered only two categories and every arc was part of the loss computation. However, it quickly became evident that neutral and avoided arcs degraded the stability of the learning loss. This is what led us to consider these four categories. Only considering part of the arcs (i.e., the positive and negative ones) for the computation of loss is referred to as *masking* in the machine learning community [16]. It helps reduce the impact of noisy data. In our case, neutral and avoided arcs are arcs from which no information is learned and for which no prediction will be asked. However, they cannot be discarded totally as they are an inherent part of the problem and must be considered for the optimization phase. This forms our first training set, based on one best-so-far solution per instance considered.

In our experiments, we also considered a second training set based on multiple best-so-far solutions. This one is an extension of the previous one. Based on the first best-so-far gathered after a first time-out, we let the solver run for a second time-out, searching for at most K solutions with the same (or better, in case it was only a best-so-far) optimum. These solutions are then aggregated by keeping the precedences present in a majority of the solution (majority defined by a given threshold percentage). This aggregated solution allows us to get rid of the potentially noisy precedences, created by some tasks that have room to move a bit within the schedule, and focus on the ones more mandatory in an optimal solution. In this training set, the neutral and avoided arcs are the same, the positive arcs are the ones presents in the majority of the solutions and the negative are the remaining ones.

In summary, to perform the learning, a specific input $x^{(i)}$ corresponds to the graph obtained from the transitive closure of a precedence graph of a solution (or aggregate solution), with features associated to each vertex, and the related target $y^{(i)}$ is a real value 0 (negative) and 1 (positive) for each unmasked pair of vertices (i.e., a possible learnable precedence link).

3.2 Step 2: Link Prediction with GNNs

Provided with training data and supervised labels, the next step is to build a function $f : P(V, E) \rightarrow [0, 1]^{V \times V}$ taking a precedence graph annotated with the node features as input and giving as output a probability for each link $(u \rightarrow v) \in V \times V$ to correspond to a precedence occurring in a high-quality solution of the instance. We designed this function using two neural networks: (1) a graph neural network computing an embedding for each node, and (2) a fully connected neural network computing a probability for each link.

First, the graph neural network is a function $f_1 : P(V, E) \to V \times \mathbb{R}^d$ which computes a d-dimensional vectorized representation for each vertex $v \in V$. The architecture is based on three SAGEConv layers (Equations (1) to (3)). The hidden dimension of each layer has a fixed size of d, and a ReLU non-linearity is used for the first two layers. The result is a graph where each node is represented by an d-dimensional embedding. We set $d = 16$.

Second, the link prediction is carried out by a function $f_2 : \mathbb{R}^d \times \mathbb{R}^d \to [0, 1]$ which computes a value for a pair of vertices $(u \to v)$. This value corresponds to the probability of this link being part of a good solution. The function is parametrized as a simple 2-layer fully-connected neural network with a ReLU activation for the hidden layer. The hidden dimension also has a size of 16. A sigmoidal transformation is finally applied to the output of the last layer to obtain a value between 0 and 1 (i.e., a probability).

Both networks are trained together with standard backpropagation using Adam optimizer [18]. Once trained, the model is used to predict new arcs from a precedence graph of an RCPSP instance. A confidence threshold $\theta \in [0, 1]$ is then used to determine whether a new precedence should be added. Let $\hat{y}^{(u \to v)}$ be the prediction obtained for a pair $(u \to v)$ that is not neutral nor avoided. For each pair we consider the arc to be positive if the prediction is greater than the threshold (i.e., $\hat{y}^{(u \to v)} \geq \theta$).

3.3 Step 3: Solving the RCPSP with New Precedences

We propose two uses of the learned preferences: (1) as new constraints in the CP model, and (2) as an information for directing the search.

First, each learned precedence (between task i and j) can be directly integrated into the model as an additional constraint ($e_i \leq s_j$). This practice is ubiquitous in constraint programming. The expected goal is to reduce the search space. It relates to streamlined constraint reasoning [14] where a partition of the search space is done using additional cutting constraints. Despite the simplicity of this approach, it comes with the critical drawback that we have no mechanism to recover from prediction errors, as adding incorrect precedence as constraints may prune the optimal solution. Consequently, the solution obtained with this first option corresponds only to an upper bound, and we lose the ability to obtain an optimality proof.

A second option is to use the new precedences to direct the search. We propose to do it by introducing an easy *ad-hoc* problem dedicated to finding an appropriate task ordering (i.e. a topological order on the precedence graph). The problem is defined as follows. A variable o_i is defined for each task i, with a domain ranging from 0 to $n - 1$. For each precedence from the instance (i.e., the neutral arcs) and each precedence learned, a constraint enforcing the precedence is added ($o_i < o_j$ if i precedes j). All the variables are also subject to an `allDifferent` constraint. Solving this problem gives an ordering satisfying all the precedences learned. By integrating randomness within the search heuristic, bias toward some solutions can be removed [33]. This ordering can then be used as a static variable ordering heuristic to solve the instance. As no additional constraint is added to the initial model, there is no risk to prune the optimal solution, and we are still able to prove optimality.

However, both options are subject to a critical concern. They are consistent only if there is no cycle obtained from the learned precedences (i.e., $a < b$, $b < c$, and $c < a$). In such a situation, the model obtained (either with additional constraints or with the ad-hoc problem) is unsatisfiable. We addressed this difficulty by adding each precedence in a graph structure maintaining the transitive closure in polynomial time. Candidate precedences are tested one by one, in order of decreasing score, and added only if they do not create a cycle. This ensures that a solution is always possible whether we add the learned precedences to the model or we use them to construct a topological order.

4 Experiments

For our experiments, we used the instances available on the PSPLIB website[2]. The best bound found so far by the community (referred by *UB* in our figures[3]) can be found on the website. PSPLIB is composed of four sizes of instances (30, 60, 90, or 120 tasks). For each size, the instances were generated by varying a given number of parameters [19] (min/max durations of the tasks, min/max number of successors, etc.). For a given set of settings, 10 instances were generated by the benchmark authors. It is thus composed of four sizes times 48 (60 for the 120 tasks) combinations of values for the parameters times 10 generations per set of parameters, yielding a total of 2040 instances. We split the instances as follows:

1. The SEEN set, composed of all but 5 of the 48 (60 for the 120 tasks) series for each of the four sizes. Among these, we select 8 instances among the 10 composing each series (1472 instances) in total. The positive and negative arcs of the best-so-far solutions of these instances are split by a k-cross-validation to train the GNN and the MLP.
2. The UNSEEN sent, composed of the remaining 2 among the 10 instances from the series selected for the SEEN set. They form the validation set as they are similar to the ones seen by the learning process but still unseen.
3. The UNKNOWN set, composed of the 5 remaining series of 10 instances for each of the four sizes (200 instances in total). They constitute the generalization set as the learning process has not seen them and has seen no other instances generated with the same set of parameters.

In some cases, we analyzed the results by size. It is then indicated within the name, i.e. UNSEEN$_{J90}$ is the instances of 90 tasks within the UNSEEN group or UNSEEN$_{\leq J60}$ for the instances of 60 tasks or less. We note that the results we report always use the same splits for comparison purposes, e.g. if a specific instance belongs to UNSEEN$_{J90}$, it is included in UNSEEN$_{J90}$ in every experiment. We also used the same split for each solution in the dataset, e.g. an instance solved with a specific timeout and options will always have the same k folds for the cross-validation.

Our neural architecture is implemented in Python using *Deep Graph Library* [37] and *Pytorch* [25]. Our experiments were run on a computing cluster equipped with a *AMD Rome 7532* CPU. For reproducibility purposes, the environment, the splits, the models, and the outputs of the runs have been stored and added to a publicly available repository[4].

4.1 Baseline CP Model for the RCPSP and Training Set

Our base CP model to solve the RCPSP is the following. For each task, the start time of task i is represented by variable[5] s_i, whose domain spans from 0 to *horizon*[6]. For each resource, we add a `cumulative` constraint. In addition, a precedence constraint ($s_i + p_i \leq s_j$

[2] `https://www.om-db.wi.tum.de/psplib/`
[3] as of March 2024
[4] `https://github.com/363734/LearningPrecRCPSP`
[5] The PSPlib files define two additional dummy tasks: 0 and $n + 1$. We do not define variables for them.
[6] We used the horizon defined within the PSPlib files.

if task i precedes task j) is added to the model for each precedence. The CP solver used is Chuffed (branch *develop*, commit *b2152f3*). The `cumulative` constraint used is the global one implemented within chuffed with the parameters `tt_filt_on`, `ttef_check_on` and `ttef_filt_on`. These parameters activate the timetabling filtering algorithm, the time-table-edge-finder check, and the time-table-edge-finder filtering algorithm [35]. The search heuristic used is *min-min-value* which corresponds to selecting the variable with the smallest minimal value and selecting its minimal value first (i.e., selecting the task that could be scheduled the earliest). We also consider two options influencing the search heuristic: `sbps` and `vsids`.

Using this model and options we generated, for each instance of PSPLIB, the best-so-far (*bsf*) solution after several timeouts (1 second, 1 minute, 10 minutes, and 1 hour) to be able to compare the influence of the quality of the dataset over the whole process. As expected, for a given fixed timeout, the solver manages to solve smaller instances to optimality. For a timeout of 1 second, the simple model (i.e., without `sbps`/`vsids`) manages to get the best *bsf* in general. The trend reverses when increasing the timeout to 10 minutes or more, where the model with `sbps` and `vsids` leads to better *bsf* solutions and more optimal solutions proven. This difference is due to the overhead of the `sbps` and `vsids` techniques. The `sbps` option alone allows to reach good solutions rapidly but has difficulties to prove optimality (`chuffed` is even warning the user when using `sbps` that it must be used with an activity-based search to optimize its efficiency). Using both options is also better at closing instances when a bigger timeout is allowed. In the rest of our experiments, we will focus on the solutions from the model with both or no options used. These best-so-far solutions are used both as training data and for comparing the methods.

4.2 Performance of the Training

We tried multiple configurations to train our model.

Training with one best-so-far per instance

Concerning the training set with one best-so-far solution per instance, we have: the timeout used to generate the dataset (1 second, 1 minute, 10 minutes or 1 hour), the options chosen (with or without `sbps`/`vsids`), and the subset of graphs chosen for training ($\text{SEEN}_{\leq J30}$, $\text{SEEN}_{\leq J60}$, $\text{SEEN}_{\leq J90}$ or $\text{SEEN}_{\leq J120}$). The learning rate is also tuned to 10^{-2}, 10^{-3} or 10^{-4}. The training is carried out for $1,000$ epochs. We performed a k-cross-validation with $k = 10$. For each of the k runs, the one with the best loss is selected. The final model is the one with the best loss among the k runs. Training takes from around $k \times 10$ min (smallest training set) to $k\times$ 1-2 hours (bigger ones). The evaluation of our training is done using the following metrics.

- Precision (*prec*): fraction of predicted positive arcs that are truly positive;
- Recall (*rec*): percentage of positive arcs correctly predicted as positive;
- True negative (*tn*): percentage of negative arcs correctly predicted negative;
- F1-score (*f1*): harmonic mean of the precision and recall.

As a general observation, there are generally more negative arcs than positive ones. Discarding some negative ones (at random) to reach equality between the two sets improves the precision of our method by a few percent (up to $3-4\%$). This technique is commonly referred to as *under-sampling*.

A preliminary analysis of the learning curves and evolution of the metrics on the training set and testing set along the learning allowed us to discard some parameters. The learning rate of 10^{-4} led to a too-slow learning and bad metrics value even after 1000 epochs compared

Table 1 Validation metrics of the two chosen models. Learning rate 10^{-2}, training set SEEN$_{\leq J120}$ generated with a timeout of 1 hour, without SBPS/VSIDS for model A, with SBPS/VSIDS for model B.

	Model A (“*without*”)				Model B (“*with*”)			
	f1	*prec*	*rec*	*tn*	*f1*	*prec*	*rec*	*tn*
SEEN$_{J120}$	0.79	0.89	0.71	0.91	0.71	0.82	0.62	0.87
UNSEEN$_{J120}$	0.78	0.89	0.70	0.92	0.71	0.83	0.62	0.87
UNKNOWN$_{J120}$	0.80	0.90	0.72	0.92	0.72	0.84	0.64	0.87
ALL$_{J120}$	0.79	0.89	0.71	0.91	0.71	0.83	0.62	0.87

Table 2 Validation metrics of models trained on smaller instances. Same training characteristic as Model B, except training set (SEEN$_{\leq J60}$ and SEEN$_{\leq J90}$). (to be compared with second column of Tab. 1).

	Model B-SEEN$_{\leq J60}$				Model B-SEEN$_{\leq J90}$			
	f1	*prec*	*rec*	*tn*	*f1*	*prec*	*rec*	*tn*
SEEN$_{J120}$	0.72	0.82	0.64	0.86	0.71	0.83	0.62	0.87
UNSEEN$_{J120}$	0.72	0.83	0.64	0.87	0.71	0.83	0.62	0.88
UNKNOWN$_{J120}$	0.73	0.83	0.65	0.87	0.73	0.83	0.64	0.87
ALL$_{J120}$	0.72	0.83	0.64	0.87	0.71	0.83	0.62	0.87

to the others. The learning rate of 10^{-2} and 10^{-3} led to very similar results. While the learning curves of 10^{-2} oscillated more, the validation metrics were generally slightly better (1% increase). The datasets generated with SBPS/VSIDS require fewer epochs to start stabilizing (i.e., having a loss close to the best loss among all the epochs) compared to the models generated on datasets without it.

Table 1 displays the result of the validation of the two most promising models. The table is organized into four rows for the evaluation of each subset of graphs (SEEN, UNSEEN, UNKNOWN and ALL) and each main column corresponds to one model. The same learning rate (10^{-2}), the same timeout (1h) for the dataset, and the same dataset (SEEN$_{\leq J120}$) are used in both models, the only distinction being the model used to create the dataset (without `sbps/vsids` for Model A and with for Model B).

In this table, we can first see that our model provides good precision but a weaker recall. In other words, the model does not make many false positives but more false negatives. This is a good prospect for our application as only predicted positive arcs lead to an impact on the model. A second observation is that the models trained with the dataset generated using the options `sbps/vsids` are globally worse than their counterparts. This is probably due to the quality of the solution. As our problem is $\mathcal{NP}$-hard [1], it is expected that learning true optimal solutions is difficult. When examining the validation results of the other models (the ones trained on benchmarks with a smaller timeout to generate the instances), we could observe the same trend as during learning, where using the 1-hour benchmarks provides better results.

We also look at the generalization capabilities of our architecture. Table 2 displays the results of two models against Model B, trained with the same characteristics as Model B, except for the instances size of the training sets, either only using the SEEN$_{\leq J60}$ or the SEEN$_{\leq J90}$. The metrics are computed for all sub-groups of different sizes. We can see that the models have good generalization capacities and have similar metrics to the one trained on SEEN$_{\leq J120}$.

Table 3 Validation metrics of Model C-SEEN$_{\leq J60}$, trained on multi-solution aggregate instances. Same training characteristic as Model C-SEEN$_{\leq J60}$, except the training set is not composed of one solution per instance, but, for each instance, of the precedences present in 70% of at most 100 solutions generated using an additional 10 minutes timeout.

	Model C-SEEN$_{\leq J60}$			
	f1	prec	rec	tn
ALL$_{J30}$	0.67	0.80	0.57	0.86
ALL$_{J60}$	0.68	0.79	0.59	0.84
ALL$_{J90}$	0.67	0.78	0.59	0.83
ALL$_{J120}$	0.72	0.83	0.64	0.87

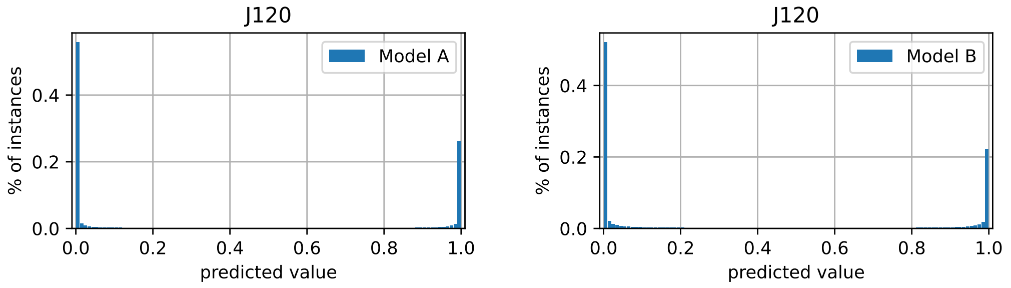
One can also notice a large gap between the metrics of smaller size compared to the j120 ones. This gap is not only present for these models with these characteristics but also present for all models trained with the "*with spbs/vsids*" training set but not in the one without. We speculate it is due to the accuracy of the solutions in the training set. In the "*with spbs/vsids*" benchmark, a majority of the solutions of instances with up to 90 tasks are optimal ones, while on the 120-task instances, it is no longer the case. Precedences within optimal solutions seem harder to predict, as the results of Table 1 were already showing.

Training with an aggregate of multiple best-so-far per instance

A natural question is to ask if choosing only one solution per instance impacts the prediction as often, there exist multiple optimal solutions for such problems. In general, there can exist multiple optimal solutions when not all the paths in the precedence graphs are critical (i.e., paths where each task starts exactly when the one preceding it ends). If the resources allow it, starting these tasks a little bit later can lead to another optimal solution. For some of these solutions, this does not change the precedences, but in some cases, it could slightly modify some of them.

To test whether our predictions had a bias while trained on a single solution for each instance, we generated, for the J30 and J60 instances, up to 100 optimal solutions (we gave an additional time out of 10 minutes to generate up to 100 solutions with the same best so far optimal value as the one found after 1 hour). From these 100 solutions, we keep only the precedences present in a majority (70%) of the solutions and trained on this subset. We thus have precedences most likely present in many solutions. We choose to focus on the smaller instances as they have shown to generalize well in training (Tab. 2). Also, as for most of them, the initial timeout of 1 hour reached the optimal solution. This training set is composed of real optimal solutions and not best-so-far solutions. Furthermore, as they are the easiest instances to solve, the 10-minute timeout allowed for the computation of 100 solutions was enough for a majority of them.

We trained a Model C-SEEN$_{\leq J60}$ on this benchmark, using the same configuration as Model B and Model B-SEEN$_{\leq J60}$ (learning rate of 0.001, 10-fold cross-validation, under-sampling and training set generated using SBPS/VSIDS active). The validation metrics (Tab. 3) are very similar to the ones present in Tab. 2. From a training point of view, there does not seem to be an impact.



(a) Edges predicted by Model A. **(b)** Edges predicted by Model B.

Figure 3 Distribution of the learned edges.

Table 4 Number of improvements/deteriorations compared to the baselines (J120 instances only).

θ	Predictions used as constraints				Predictions used for ordering			
	to=1s	to=1m	to=10m	to=1h	to=1s	to=1m	to=10m	to=1h
	Predictions from Model A used on a model without SBPS/VSIDS							
0.5	0/600	0/600	0/600	0/600	24/391	4/373	6/360	8/349
0.55	0/600	0/600	0/600	0/600	28/387	4/367	2/358	7/348
0.75	0/599	0/600	0/599	1/598	26/394	4/370	4/362	3/357
0.95	1/585	0/584	2/584	3/582	22/405	5/372	9/357	8/350
0.99	4/559	6/557	5/554	4/553	27/399	6/371	5/359	7/350
	Predictions from Model B used on a model without SBPS/VSIDS							
0.5	0/600	0/600	0/599	0/599	31/392	7/371	7/365	7/355
0.55	0/600	0/599	1/599	0/598	21/404	6/367	6/363	9/360
0.75	1/597	0/597	2/594	3/594	31/388	6/372	7/358	7/352
0.95	5/554	4/556	2/553	3/550	23/404	9/365	11/357	9/349
0.99	21/487	5/497	7/495	6/493	25/393	7/373	5/362	8/356
	Predictions from Model A used a model with SBPS/VSIDS							
0.5	480/94	124/476	7/593	4/596	439/0	185/6	5/0	1/0
0.55	483/98	122/478	9/590	7/592	440/11	188/5	6/3	1/3
0.75	470/114	116/480	13/586	12/587	431/3	186/5	7/4	2/5
0.95	416/163	106/474	17/564	24/561	413/50	166/50	6/20	2/12
0.99	411/163	108/454	24/533	28/526	413/45	172/39	6/25	1/13
	Predictions from Model B used a model with SBPS/VSIDS							
0.5	475/110	116/484	10/590	8/590	445/0	186/4	5/1	2/2
0.55	447/127	116/483	11/588	10/590	421/25	176/19	7/7	2/4
0.75	457/113	114/481	15/580	14/577	454/0	191/1	7/1	2/2
0.95	413/159	111/449	26/525	21/532	433/26	179/27	6/14	2/8
0.99	419/152	125/386	26/469	34/459	433/26	179/25	8/17	2/8

4.3 Performance for Solving the RCPSPs

We decided to focus the presentation of our results on the ALL_{J120} as it contains the most open instances and thus the most interesting. We analyzed the results split among the SEEN_{J120}, UNSEEN_{J120}, and UNKNOWN_{J120} and did not find a significant difference between them, allowing us to confirm no sign of overfitting, as hinted by Tab. 1, hence why we present the results on ALL_{J120} only.

Distribution of predictions

We analyzed first the distribution of the prediction done among the edges of the J120 case in Fig. 3, for both Model A (Fig. 3a) and Model B (Fig. 3b). We can see that our process is confident on the outcome for most of the edges. Normally, we select as a positive prediction

Table 5 Generalization ability (to be compared with the last line of Tab. 4).

θ	Model B-SEEN$_{\leq J60}$							
	1s	1m	10m	1h	1s	1m	10m	1h
	Predictions used as constraints				Predictions used for ordering			
0.99	421/141	119/398	27/484	35/481	430/17	187/18	6/10	1/4

the precedence with a prediction score of at least 0.5. Given the distribution of the edges and the fact that the recall score is not that high, we would want to verify if increasing the threshold of the selected prediction could impact the final results. To this end, we decided to test multiple thresholds θ: 0.5, 0.55, 0.75, 0.95, and 0.99.

Comparison to Baseline

We run Model A and Model B to predict the edges of the full J120 benchmark. These predictions are then filtered given the threshold and then used either as *additional constraints* or to create an *ordering-based heuristic*, both to be used with the model with the SBPS/VSIDS options or without the options (Tab. 4). The tables gather, for each threshold, each timeout, each usage, and each generative model, the number of instances among the 600 where there is an improvement/deterioration of the bound compared to the baseline (i.e., the model with the same options used and the same timeout but without the learned approach).

Multiple observations can be made by comparing these numbers. First, using our ordering-based technique without a SBPS/VSIDS heuristics does not provide good results. This can be explained by the fact it has already been observed that static ordering performs generally slower than dynamic ones [27], allowing an easier deterioration of the bound. Another observation is that using precedence as an additional constraint leads to many deterioration. This is a logical consequence of the fact that adding new constraints creates a restriction and thus, if one of these constraints is wrong, removes the optimum. We can see that increasing the threshold mitigates this effect, as expected. Solutions based on the use of prediction from Model A are also generally a bit less good than when using Model B. A model trained on a more qualitative training set is thus preferable.

Interestingly, we can also notice that with shorter timeouts, using the ordering-based method manages to improve more bounds, as it can guide the search towards better solutions first. The restriction-based one improves more bounds with bigger timeouts, as by reducing the search space, it can potentially reach part of the search space that was not reached before within the timeout. This, however, works only if the restriction does not select bad precedences, as shown by the important number of deteriorations.

Generalization Ability

Figure 5 compares the results of Model B against Model B-SEEN$_{\leq J60}$. Our validation results on this model show that it had similar accuracy to Model B. To confirm it, we made the predictions on all J120 instances and solved them using the model with SBPS/VSIDS. Our results show that it generalizes very well as the results are comparable to the ones of Model B with the same other parameters (last line of Tab. 4).

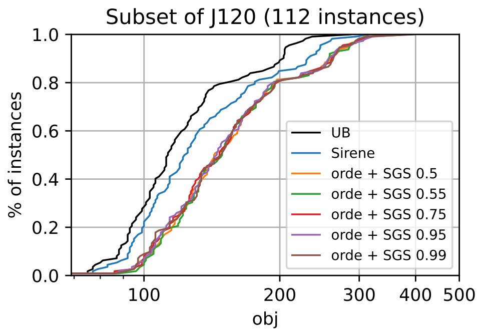


Figure 4 SGS-based solution.

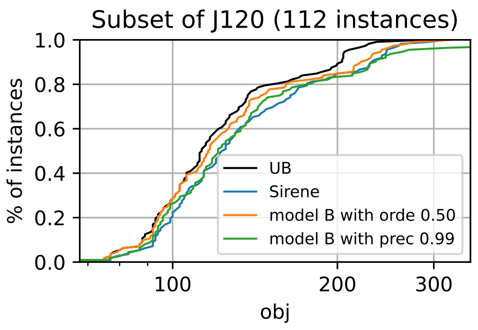


Figure 5 Comparison with SIRENE [31].

Schedule Generation Scheme (SGS)

A third way to use the prediction is to use an SGS, as done in [31]. Given an ordered set of tasks, the SGS greedily constructs a schedule, ensuring the resources and precedences constraints are respected. Fed with an ordering corresponding to the optimal solution, the SGS can reconstruct the optimal solution. Figure 4 displays, in a cactus plot, the bound of the solutions found after applying the SGS on our ordering produced by the prediction of Model B with the various thresholds. We compare to the SIRENE algorithm [31], using the same SGS. For the comparison, we used the results stored on their public repository. These bounds are available for 408 instances among the PSPLIB benchmark and were used to generate Figures 4 and 5 of their paper. Figure 4 focuses on the J120 among these (112 among the 600). The discrepancy between their method and our use of the SGS can be explained by the fact that their method works on a global view. By extracting an ordering from a close-to-a-solution, they guarantee that if some part of the schedule is right, it will stay the same after using the SGS. In our case, it only works if there are no errors within the predicted arcs. Observing that there is not much improvement in the solutions after using the SGS on our ordering with different thresholds confirms that there are predictive errors even when the learning process is sure of its prediction.

Comparison to Sirene

SIRENE [31] is the closest approach also using GNN found in the literature. While the two methods are very close to each other, they are in fact complementary. SIRENE focuses on a more global type of prediction (i.e., a potential solution to be corrected) while we focus on a more local prediction (i.e., new precedence between tasks to be used). Another conclusion we share is that improving the bounds obtained in our training set is difficult. In their case, they improve drastically the runtime while obtaining no bound improvement (Fig. 5) compared to their baseline. In our case, we managed to improve a few bounds (Tab. 4), and especially improve the quality of early solutions. It confirms the potential of GNNs to replicate statistical distribution (here the distribution of solutions provided in our training sets) but not to be able to solve the problem given and reason on it. Another limitation compared to SIRENE is that our current approach as our model is targeted to 4-resource problems. In our opinion, for instances with more features, we think an aggregation of multiple predictions for multiple sub-problems defined by selecting four features of the instance is a possible solution to get around this limitation. Fig. 5 shows a cactus plot, comparing the best-known upper bound (*UB*) to the makespan (from PSPLIB website), SIRENE, the predictions of Model

Table 6 Results when training on an aggregate of multiple solutions (to be compared with Fig.5).

θ	Model C-SEEN$_{\leq J60}$							
	1s	1m	10m	1h	1s	1m	10m	1h
	Predictions used as constraints				Predictions used for ordering			
0.99	436/108	154/332	46/427	48/424	451/0	193/1	7/2	0/2

B with $\theta = 0.5$ used for ordering on a model using Sbps/Vsids and time of 1 hour and the predictions of Model B with $\theta = 0.99$ used for additional constraint on a model using Sbps/Vsids and timeout of 1 hour.

Impact of multiplicity of optimal solution

Table 6 shows the result when Model C-SEEN$_{\leq J60}$ is used to predict precedences on the J120 instances. We also used them either in addition to the model either to create an ordering, both being used with the Sbps/Vsids generic heuristics when solving. When comparing to Fig. 5, one can see that using an aggregate of multiple solutions can help to improve the results. This comes from the fact that the precedences learned are the ones present in many optimal solution at once thus directing to any of these solutions the same way.

However, the drawback of this training set is that it requires multiple solutions. By using the generalization abilities of our model we were able to keep the additional computations cost relatively low. However, doing the same with bigger instances in the training set (by training on SEEN$_{\leq J120}$ like Model A and Model B) would be intractable as for many of the bigger instances, generating one good solution sometimes takes up to one hour, generating several of them would then be too tedious.

5 Conclusion and Perspective

This paper proposed a novel approach based on graph neural networks to predict new precedences for the resource-constrained project scheduling problem. The learned precedences can then be used either as additional constraints to get a stronger filtering or as a heuristic to drive the search. A high precision in the precedences learned has been obtained after the training. Our experiment on the PSPLIB benchmark confirms that, due to the $\mathcal{NP}$-hardness of the problem, a high recall is difficult to reach, but that we can nevertheless speed up the solving process when using a dynamic ordering. An improvement of our baseline (i.e., best-so-far after given time-out) has been observed in our experiment but remains difficult to achieve. The quality of the prediction depends on the quality of the training set. Using aggregates of multiple solutions allows learning of more crucial precedences. Our experiments shows also a good generalization as models trained on instances with less or equal than 60 tasks can achieved similar results on instances with 120 tasks as models trained on instances with less or equal than 120 tasks. Our method is solver agnostic and could even be combined with other metaheuristics such as a large neighborhood search.

One of our perspectives is to study whether the learned precedences are dependent on the training benchmark. Another is to reflect on how we can make predictions on instances with a different number of resources than trained for. Among our perspectives is also to apply this methodology to other variants of scheduling problems such as the RCPSP with time windows and the job shop scheduling problem. We also expect this could generalize to other combinatorial problems with an underlying graph structure such as job-shop scheduling problems.

References

1 Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete applied mathematics*, 5(1):11–24, 1983.

2 Lei Cai, Jundong Li, Jie Wang, and Shuiwang Ji. Line graph neural networks for link prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5103–5113, 2021.

3 Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

4 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

5 Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 392–409. Springer, 2021.

6 Mark Cheung. *Geometric Deep Learning: Impact of Graph Structure on Graph Neural Networks*. PhD thesis, Carnegie Mellon University, 2022.

7 Geoffrey Chu and Peter J Stuckey. Learning value heuristics for constraint programming. In *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*, pages 108–123. Springer, 2015.

8 Bert De Reyck et al. A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 111(1):152–174, 1998.

9 Emir Demirović, Geoffrey Chu, and Peter J Stuckey. Solution-based phase saving for cp: A value-selection heuristic to simulate local search behavior in complete solvers. In *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 99–108. Springer, 2018.

10 Floris Doolaard and Neil Yorke-Smith. Online learning of variable ordering heuristics for constraint optimisation problems. *Annals of Mathematics and Artificial Intelligence*, pages 1–30, 2022.

11 Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893*, 2019.

12 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.

13 Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

14 Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In *International Conference on Principles and Practice of Constraint Programming*, pages 274–289. Springer, 2004.

15 Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

16 Bo Han, Jiangchao Yao, Gang Niu, Mingyuan Zhou, Ivor Tsang, Ya Zhang, and Masashi Sugiyama. Masking: A new perspective of noisy supervision. *Advances in neural information processing systems*, 31, 2018.

17 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

18 Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

19 Rainer Kolisch and Arno Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European journal of operational research*, 96(1):205–216, 1997.

20 Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

21 Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *International Conference on Learning Representations*, 2016.

22 Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications*, 390(6):1150–1170, 2011.

23 Tom Marty, Tristan François, Pierre Tessier, Louis Gautier, Louis-Martin Rousseau, and Quentin Cappart. Learning a generic value-selection heuristic inside a constraint programming solver. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

24 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

25 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

26 A Alan B Pritsker, Lawrence J Waiters, and Philip M Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management science*, 16(1):93–108, 1969.

27 Patrick Prosser. The dynamics of dynamic variable ordering heuristics. In *Principles and Practice of Constraint Programming—CP98: 4th International Conference, CP98 Pisa, Italy, October 26–30, 1998 Proceedings 4*, pages 17–23. Springer, 1998.

28 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

29 Andreas Schutt, Thibaut Feydy, and Peter J Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10*, pages 234–250. Springer, 2013.

30 Wen Song, Zhiguang Cao, Jie Zhang, Chi Xu, and Andrew Lim. Learning variable ordering heuristics for solving constraint satisfaction problems. *Engineering Applications of Artificial Intelligence*, 109:104603, 2022.

31 Florent Teichteil-Königsbuch, Guillaume Povéda, Guillermo González de Garibay Barba, Tim Luchterhand, and Sylvie Thiébaux. Fast and robust resource-constrained scheduling with graph neural networks. In Sven Koenig, Roni Stern, and Mauro Vallati, editors, *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*, pages 623–633. AAAI Press, 2023. `doi:10.1609/ICAPS.V33I1.27244`.

32 Ronald van Driel, Emir Demirović, and Neil Yorke-Smith. Learning variable activity initialisation for lazy clause generation solvers. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 62–71. Springer, 2021.

33 Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution sampling with random table constraints. *Constraints*, pages 1–33, 2022.

34 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

35 Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings 8*, pages 230–245. Springer, 2011.

36 Julien Vion and Sylvain Piechowiak. Une simple heuristique pour rapprocher dfs et lns pour les cop. *Proceedings of JFPC'17*, pages 39–45, 2017.

37 Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

38 Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, and Le Song. *Graph neural networks*. Springer, 2022.

39 Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.