Learning a Generic Value-Selection Heuristic Inside a Constraint Programming Solver

Tom Marty ✉ 🏠 📛
Polytechnique Montréal, Montreal, Canada
Ecole Polytechnique, Palaiseau, France

Tristan François ✉
Ecole Polytechnique, Palaiseau, France

Pierre Tessier ✉
Ecole Polytechnique, Palaiseau, France

Louis Gautier ✉
Ecole Polytechnique, Palaiseau, France

Louis-Martin Rousseau ✉ 🏠 📛
Polytechnique Montréal, Montreal, Canada

Quentin Cappart ✉ 🏠 📛
Polytechnique Montréal, Montreal, Canada

—— **Abstract** ——

Constraint programming is known for being an efficient approach to solving combinatorial problems. Important design choices in a solver are the *branching heuristics*, designed to lead the search to the best solutions in a minimum amount of time. However, developing these heuristics is a time-consuming process that requires problem-specific expertise. This observation has motivated many efforts to use machine learning to automatically learn efficient heuristics without expert intervention. Although several generic *variable-selection heuristics* are available in the literature, the options for *value-selection heuristics* are more scarce. We propose to tackle this issue by introducing a generic learning procedure that can be used to obtain a value-selection heuristic inside a constraint programming solver. This has been achieved thanks to the combination of a *deep Q-learning* algorithm, a tailored *reward signal*, and a *heterogeneous graph neural network*. Experiments on *graph coloring*, *maximum independent set*, and *maximum cut* problems show that this framework competes with the well-known impact-based and activity-based search heuristics and can find solutions close to optimality without requiring a large number of backtracks.

1   Introduction

Combinatorial optimization has countless industrial applications, such as scheduling, routing, or finance. Unfortunately, most of these problems are NP-hard and, thereby, challenging to solve efficiently. It is why finding good solutions has motivated intense research efforts for many years. Traditional methods for tackling them are somehow based on a *search procedure*: A clever enumeration of the solution space is performed to find a feasible and possibly optimal solution. Among these methods, *constraint programming* (CP) is an exact procedure. It constitutes a popular approach as it offers the possibility to find the optimal solution or good feasible approximations by stopping the search early. An additional asset is its declarative paradigm in modeling, which makes the technology easier for the end-user to grasp. Introducing solver-agnostic modeling languages, such as MiniZinc [35] has greatly facilitated this aspect. Aligned with this goal, the propagation engine inside a CP solver is mostly hidden from the end-user. However, ensuring a generic search procedure is trickier as non-trivial heuristics must be designed to make the solving process efficient for an arbitrary problem. That being said, generic *variable-selection* and *value-selection* heuristics have been successfully designed. Notable examples are *impact-based search* [37] or *activity-based search* [31], but they require computationally intensive initialization and yield poor performance at the beginning of the search. This makes these methods not always appropriate for general use. As a concrete example, the current version of MiniZinc[1] does not propose generic value-selection heuristics, except *in(out)domain* or *impact-based search.* In practice, heuristics are often designed thanks to problem-specific expert knowledge, which is often out of reach for end-users that do not have a solid background in artificial intelligence.

In another context, *machine learning* (ML) has been recently considered for automating the design of branching heuristics, both in constraint programming [11], mixed-integer programming [16, 23], or SAT solving [41]. Specifically, *reinforcement learning* (RL) [45] or *imitation learning* [22] approaches, often combined with *deep learning* [27], have gained special attention. Although this idea seems appealing, this is not an easy task to achieve in practice as several technical considerations must be taken into account in order to ensure both the efficiency and the genericity of the approach. In constraint programming, we identified three questions to resolve when learning a generic branching heuristic inside a solver. They are as follows:

1. *How to train the machine learning model?* An intuitive way is to leverage an RL agent that would explore the tree search by making branching decisions and rewarding it based on the quality of the solution found on a terminal node. This would typically be done with a *depth-first search* traversal of the tree for getting a certificate of optimality. However, as pointed out by several authors [38, 42], the backtracking operations inside a solver raise difficulties when formalizing the task as a Markov decision process and may require redefining it. Besides, this training scheme intensifies the *credit assignment problem* [32], ubiquitous in reinforcement learning.

2. *How to evaluate the quality of a value selection?* A core component of an RL environment is the *reward function*, which gives a score to each decision performed. The end goal for the agent is to perform a sequence of decisions leading to the best-accumulated sum of rewards. In our case, an intuitive solution would be to reward the agent according to the quality of the solution found. However, this information is only available at terminal nodes, and only a zero reward is provided in branching nodes. This is related to the *sparse reward* problematic, which is known to complicate the training process.

---

[1] `https://www.minizinc.org/doc-2.7.0/en`

3. *How to learn from a CP model?* This question relates to the type of architecture that can obtain a value-selection heuristic from a search node (i.e., a partially solved CP model). A promising direction has been proposed by Gasse et al. [16] for binary mixed-integer programs. They introduced a bipartite graph linking variables and constraints (i.e., the two types of nodes) when a variable is involved in a given constraint. The subsequent architecture is a *heterogeneous graph neural network*. However, this encoding is not directly applicable in constraint programming, as a CP model generally involves non-binary variables and combinatorial constraints. This has been partially addressed by Chalumeau et al. [12], who introduced a tripartite graph where variables, values, and constraints are specific types of nodes. However, this approach lacks genericity as the method requires retraining when the number of variables changes.

To our knowledge, answering such questions is still an open challenge in the research community. This paper proposes to progress in this direction. It introduces a generic learning procedure that can be used to obtain a value-selection heuristic from a constraint programming model given as input. The approach has been designed to be generic in that it can be used for any CP model given as input. In practice, a specific way to extract *features* from a constraint should be designed for any available constraint, but this has to be done only once per constraint type. In this proof of concept, we limit our experiments to three combinatorial optimization problems, namely *graph coloring*, *maximum independent set*, and *maximum cut*. Specifically, we propose three main contributions, each dedicated to addressing one of the aforementioned difficulties. They are as follows: (1) a learning procedure, based on restarts, for training a reinforcement learning agent directly inside a CP solver, (2) a reward function able to assign non-zero intermediate rewards based on the propagation that has been carried out on the node, and (3) a neural architecture based on a tripartite graph representation and a heterogeneous graph neural network. Experimental results show that combining these three ideas enables the search to find good solutions without requiring many backtracks and competes with the well-known impact-based and activity-based search heuristics.

The paper is structured as follows. The next section presents other approaches related to our contribution. Then, Section 3 introduces succinctly technical background on reinforcement learning and graph neural networks. The core contributions are then presented in Section 4. Finally, Section 5 provides experimental results and closes with a discussion of the results.

2 Related Work

Bengio et al. [5] identified three ways to leverage machine learning for combinatorial optimization. First, *end-to-end* learning aims to solve the problem only with a trained ML model. This has been, for instance, considered for the traveling salesman problem [4, 25]. However, such an approach does not guarantee the validity nor optimality of the solution obtained. Second, *learning to configure* is dedicated to providing insights to a solver before its execution. This can be, for instance, the decision to linearize the problem in the context of quadratic programs [7] or to learn when a decomposition is appropriate [26]. This approach is also referred to as parameter tuning [21]. We refer to the initial survey for extended information about these two families of approaches. Third, *learning within a search procedure* uses machine learning within the solver. Our contribution belongs to this last category of methods. Although the idea of combining learning and searching for solving combinatorial optimization problems was already discussed in the nineties [36], it has re-emerged recently with the rise of deep learning. Most combinatorial optimization solvers are based on *branch-and-bound*

and *backtracking*. In this context, ML is often used with branching rules to follow. *Imitation learning* [22] has been for instance used to replicate the expensive *strong branching* strategy for mixed-integer programming solvers [16, 23]. One limitation of imitation learning is that the performances are bounded by the performance of the imitated strategy, which remains heuristic and perfectible [43]. This opens the door for RL approaches that have the guarantee to find the best branching strategy eventually [29]. A branching strategy can be split into two challenging decisions, *variable-selection* and *value-selection*. Reinforcement learning approaches have been considered for both of them.

Concerning the learning for selecting the next variable to branch on, Song et al. [42] proposed to combine a *double deep Q-network* algorithm [49] with a graph neural network for carrying out this task. The approach is trained to minimize the expected number of nodes to reach a leaf node using the *first-fail principle*. Although this is a good proxy for pruning a maximum of infeasible solutions for a constraint satisfaction problem, it does not extend naturally to optimization variants, for which one should consider a trade-off between the quality of the solution found and the number of nodes required to reach that solution. Similarly, van Driel et al. [48] leveraged a graph neural network to initialize a variable-selection heuristic for *Chuffed*, a hybrid CP-SAT solver. In an online setting, Doolard and Yorke-Smith [15] also proposed to learn variable ordering heuristics where training time is included in the total solving time. Bandit-based learning approaches were also considered by Xia and Yap to automatically select search heuristics [50].

For the value-selection heuristic, Chu and Stuckey [13] introduced a scoring function which gives a score indicating how good an assignation is, given the current domain. A training phase is the carried out in a supervised manner to learn this scoring function. Cappart et al. [11] proposed to train a model with reinforcement learning outside the CP solver and to integrate the agent, once trained, subsequently in the solver. This has been achieved by reaping the benefits of a dynamic programming formulation of a combinatorial problem. An important limitation of this work is that no information related to the CP solver, such as the propagation achieved on a node, can be used to drive the decision. Chalumeau et al. [12] mitigated this issue by carrying out the learning inside the solver. The model is trained to find the optimal solution and to prove it with the least number of explored search nodes. However, this goal is disconnected from finding the best solution as quickly as possible and is practically hard to achieve, even with a good heuristic. A more realistic goal is to find a good solution quickly without closing the search. This is how the contribution of this paper is positioned.

We want to point out that learning *how to branch* is not the only way to leverage ML inside a combinatorial optimization solver. Related works have also been proposed on learning tight optimization bounds [9] or for accelerating column generation approaches [34]. A recurrent design choice is an architecture based on graph neural networks. We refer to the following survey for more information about combinatorial optimization with graph neural networks [10].

3   Technical Background

This section introduces the required background on reinforcement learning and graph neural network to grasp the technical aspects of the paper.

3.1 Reinforcement Learning

Let $\langle S, A, T, R \rangle$ be a 4-tuple representing a *Markov decision process* where S is the set of states in the environment, A is the set of actions that the agent can do, $T : S \times A \to S$ is a transition function leading the agent from one state to another, given the action taken, and $R : S \times A \to \mathbb{R}$ is a reward function of taking an action from a specific state. The sequence $[s_1, \ldots, s_T]$ from the initial state (s_1) of an agent towards a terminal state (s_T) is referred to as an *episode*. The returned reward within a partial episode $[s_t, \ldots, s_T]$ can be formalized as follows: $G_t = \sum_{i=t}^{T} R(s_i, a_i)$. We intentionally omitted the discounting factor as we do not want to discount the late rewards in our application. The agent is governed by a policy $\pi : S \to A$, which indicates the action that must be taken on a given state. The agent's goal is to find the policy that will lead it to maximize the accumulated reward until a terminal state is reached. The core idea of reinforcement learning is to determine this policy by letting the agent interact with the environment and increasing the probability of taking action if it leads to high subsequent rewards. There are a plethora of reinforcement learning algorithms dedicated to this task, such as *trust region policy optimization* [40] or *soft actor-critic* [18]. We refer to *SpinningUp* website for explanations of the main algorithms [1].

This section presents the core principles of *deep Q-learning* [33], which is the algorithm used in this paper. The idea is to compute an *action-value* function $Q^\pi(s_t, a_t) = G_t$. Intuitively, this function gives the accumulated reward that the agent will obtain when performing the action a at state s while subsequently following a policy π. The output of this function for a specific action is referred to as a *Q-value*. Provided that the action-value function can be computed exactly, the optimal policy $\pi^\star$ turns to be simply the selection of the action having the highest Q-value on a specific state: $\pi^* = \mathsf{argmax}_\pi Q^\pi(s, a), \; \forall (s, a) \in (S, A)$. Although the exact computation of Q-values can theoretically be performed, a specific value must be computed for each pair of states and actions, which is not tractable for realistic situations. It is why a tremendous amount of work has been carried out to approximate accurately and efficiently Q-values. Among them, deep Q-learning aims to provide a neural estimator $\hat{Q}(s, a, \theta) \approx Q(s, a)$, where θ is a tensor of parameters that must be learned during a training phase. This algorithm is commonly enriched with other mechanisms dedicated to speed-up or stabilizing the training process, such as the *double deep Q-network* variant [49] or *prioritized experience replay* [39]. Concerning the neural architecture, we opted for a graph neural network, which is explained in the next section.

3.2 Graph Neural Network

Intuitively, the goal of a *graph neural network* (GNN) is to embed information contained in a graph (e.g., the structure of the graph, spatial properties, features of the nodes, etc.) into a d-dimensional tensor for each node $u \in V$ of the graph. To do so, information on a node is iteratively refined by aggregating information from neighboring nodes. Each iteration of aggregation is referred to as a *layer* of the GNN and involves parameters that must be learned. Let $h_u^k \in \mathbb{R}^{d \times 1}$ be the tensor representation of node u at layer k of the GNN, $h_u^{k+1} \in \mathbb{R}^{l \times 1}$ be the tensor representation of this node at the next layer (l being the dimension of a node at the layer $k+1$), and $\theta_1 \in \mathbb{R}^{l \times d}$ and $\theta_2 \in \mathbb{R}^{l \times d}$ be two matrices of parameters, respectively. Each GNN layer carries out the following update:

$$h_u^{k+1} = g\left(\theta_1 h_u^k \; \star \; \Big(\bigoplus_{v \in N(u)} \theta_2 h_v^k \Big) \right) \; \forall u \in V \tag{1}$$

Three operations are involved in this update: (1) $\bigoplus$ is an *aggregation* operator that is dedicated to aggregating the information of neighbors (e.g., *mean-pooling* or *sum-pooling*), (2) $\star$ is a *merging* which enables to combine of the information of a node with the ones from the

neighbors (e.g., a concatenation), and (3) g is an element-wise non-linear activation function, such as the ones commonly used in fully-connected neural networks (e.g., ReLU [17]). Without loss of generality, the *bias* term is not included in the equation. A concrete implementation of a GNN defines these three functions adequately. The training is conducted in a fully-connected neural network through back-propagation and an optimizer based on stochastic gradient descent such as Adam [24].

## 4   Learning a Value-Selection Heuristic Inside a Solver

This section presents how a value-selection heuristic can be learned with reinforcement learning in a CP solver from a model given as input. This is the core contribution of the paper. Three mechanisms are introduced: (1) *a training procedure based on restarts*, (2) *a reward function leveraging propagation of domains*, and (3) *a heterogeneous graph neural network architecture.* They are described individually in the next subsections. They have been implemented in the recently introduced *SeaPearl.jl* solver [12]. Inspired by the architecture of MiniCP [30], the main specificity of SeaPearl is to natively integrate support for learning inside the search procedure. This greatly facilitates the prototyping of new search algorithms based on learning.

4.1   Restart-Based Training

Generally speaking, the performance of a reinforcement learning agent is tightly correlated with the definition of an *episode.* This corresponds to the agent's interactions with the CP solver's search procedure and is related to the goal desired for the agent. Two options are discussed in this section, (1) an *episode based on depth-first search*, introduced by Chalumeau et al. [12], and (2) an *episode based on restarts*, which is our first contribution.

Building branching heuristics for solving exact combinatorial optimization problems often concurrently targets two objectives: *finding quickly good solutions* and *proving the optimality of a solution.* The approach of Chalumeau et al. [12] relies heavily on the second objective and aims to minimize the number of visited search nodes before proving optimality (e.g., closing the search). To do so, they defined a training episode as a complete solving process carried out by the depth-first search of a solver and penalized through the reward function the generation of each node. This is illustrated in the left picture of Figure 1. However, this approach suffers from an important difficulty. An episode only terminates when the search is completed, which is often intractable for realistic problems as it requires exploring an exponentially large search tree. This is especially problematic during training, where the heuristic is still mediocre. In addition, using a depth-first search algorithm in a Markov Decision Process (MDP) framework required additional considerations not considered by Chalumeau et al. [12]. For example, using a backtracking algorithm in a regular temporal MDP renders their method prone to the *credit assignment problem* [32]. These considerations have been pointed out by Scavuzzo et al. [38] for mixed-integer programming.

Unlike this approach, we propose to train the model to find high-quality solutions quickly. To do so, we followed the approach proposed by Cappart et al. [11]: an episode is defined as a *single dive* in the search tree. No backtrack is allowed; the episode stops when a complete solution is found or a failure is generated. Once the episode is terminated, a restart from the root node is performed, and a new episode is generated, hence the name of *restart-based episode.* This is illustrated in the right picture of Figure 1. One limitation of Cappart et al. [11] is that episodes are executed outside the CP solver during the training and cannot use the information updated during propagation for the branching. Inspired by Song et
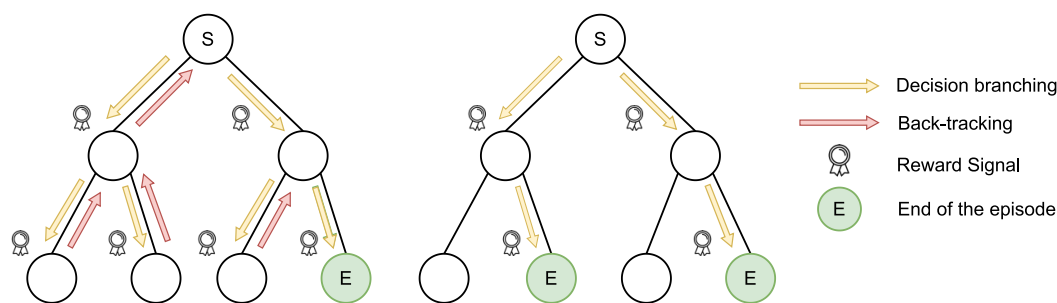
Figure 1 The two training procedures (left: *depth-first search* [12], right: *restart-based* - ours).

al. [42] for variable-selection heuristics, we addressed this limitation by executing each episode inside the solver during the training. Formally, this requires defining the dynamics of the environment as a Markov Decision Process (i.e., a tuple $\langle S, A, T, R \rangle$, see Section 3.1). It is defined as follows.

Set of states Let $\mathcal{P} = \langle X, D(X), C, O \rangle$ be the expression of a combinatorial optimization problem (COP), defined by its variables (X), the related domains (D), its constraints (C), and an objective function (O). Each state $s_t \in S$ is defined as the pair $s_t = (\mathcal{P}_t, x_t)$, where $\mathcal{P}_t$ is a partially solved COP (i.e., some variables may have been assigned), and $x_t \in X$ is a variable selected for branching, at step t of the episode. The initial state $s_1 \in S$ corresponds to the situation after the execution of the fix-point at the root node. A terminal node is reached either if all the variables are assigned ($\forall x \in X : |D_t(x)| = 1$), or if a failure is detected ($\exists x \in X : |D_t(x)| = 0$). The variable selected for branching is obtained through a standard heuristic such as *first-fail*.

Set of actions Given a state $s_t = (\mathcal{P}_t, x_t)$, an action a_t corresponds to the selection of a value $v \in D(x_t)$ for branching at step t. Finding the most promising value to branch on is the problem addressed in this paper.

Transition function Given a state $s_t = (\mathcal{P}_t, x_t)$ and an action $a_t = v$, the transition function executes three successive operations. First, it assigns the value v to the variable x (i.e., $D(x_{t+1}) = v$). Second, it executes the fix-point on $\mathcal{P}_t$ in order to prune the domains (i.e., $\mathcal{P}_{t+1} = \mathsf{fixPoint}(P_t)$). Third, it selects the next variable to branch on (i.e., $x_{t+1} = \mathsf{nextVariable}(P_{t+1})$). This results in a new state $s_{t+1} = (\mathcal{P}_{t+1}, x_{t+1})$. Integrating the propagation inside the transition is one important difference with Cappart et al. [11].

Reward function The function is defined separately in Section 4.2.

Concerning the training, we opted for a *double deep Q-learning* algorithm [49], known to perform well for discrete action spaces. However, other RL algorithms could also be used. We compared our restart-based training procedure using a simple terminal reward based on the solution's score with the backtracking-based approach of Chalumeau et al. [12] using their reward at each step (penalty of 1 for each explored node). We selected the *maximum independent set problem* for this comparison with instances with 50 nodes. Results are presented using performance profiles [14] in Figure 2. A detailed explanation of the experimental protocol is proposed in Section 5.

We evaluated both methods on two metrics matching the objective for which they were specifically trained. We look at the value of the solution obtained after a single dive (Figure 2a) in the tree search and the number of nodes visited to prove optimality using a depth-first search (Figure 2b). As expected, we observe that the agent trained with the *restart-based learning* strategy allows good results regarding the optimality gap for the first solution found

after a single dive. Remarkably, our method yields a comparable ability to prove optimality compared to Chalumeau et al. [12], whose primary aim was specifically to solve the problem in the minimum number of nodes. This last result has to be mitigated as both RL-based methods lie in the range of the random strategy (shaded blue area).
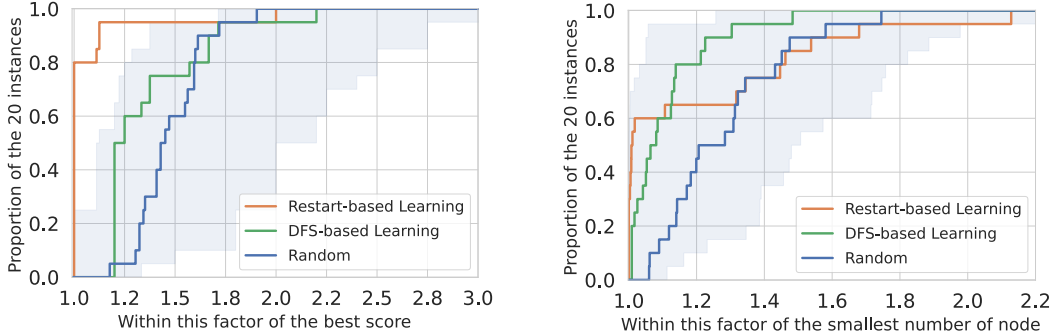
(a) Score of the first solution obtained.   **(b)** Number of node visited until optimality.

■ **Figure 2** Comparison of both training methods on maximum independent set (50 nodes). As a non-learned baseline, we added the performances of an agent performing only random decisions. Training is carried out on randomly generated Barabási-Albert graphs [2]; we selected this type of distribution as the generated graphs are known to mimic human-made and natural organizations. The evaluation is performed on 20 other graphs following the same distribution.

Finally, as shown in Figure 2a, it is important to notice that the optimality gap returned by our method is still non-negligible at the first solution obtained. The complexity of a combinatorial problem lies mainly in closing this gap, which is why backtracking is required. Experiments with backtracking are proposed in Section 5.

4.2   Propagation-Based Reward

The definition of our reward must be aligned with our objective of *finding quickly good solutions* for the combinatorial problem. Based on our training procedure, an intuitive function is to reward the agent proportionally to the solution quality found at the end of an episode. In case of an infeasible solution found, a penalty can be given. The main drawback of this rewarding scheme is that this information is only available at terminal nodes, and no reward is provided in branching nodes. This is related to the *sparse reward* problem, which complicates the training process [47]. To address this challenge, one should find a way to give informative intermediate rewards along the solving process. To this end, we propose a new rewarding scheme based on the domain reduction of the objective variable (i.e., the variable that must be minimized or maximized). This reduction happens either thanks to the branching assignment or the application of the fix-point. There are two main components: (1) an *intermediate reward* (r^{mid}) collected at branching nodes, and (2) *terminal reward* (r^{end}) collected only at the end of an episode.

Assuming a minimization problem, the intermediate reward follows two principles: each domain reduction of the largest values of the domain is rewarded, and each domain reduction of the lowest values of the domain is penalized. It is important to note that following these principles does not guarantee the discovery of a good solution at the end of the branch. The rationale is to lead the agent to a situation where the minimum cost can be *eventually* obtained while removing costly solutions. It is formalized in Equations (2) to (4), where r^{mid}_t is the reward obtained at step t, and is illustrated in Figure 3. As shown in Equation (5),

the terminal reward is set to -1 if the leaf node corresponds to an infeasible solution and 0 if it is feasible. Finally, the total reward (r^{acc}) accumulated during an episode of T steps is the sum of all intermediate rewards with the final term, as proposed in Equation (6).
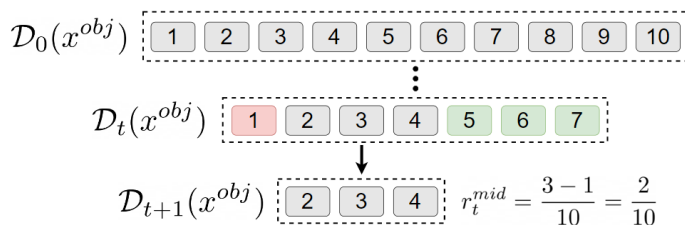
$$r_t^{\mathsf{ub}} = \#\left\{v \in D_t(x^{\mathsf{obj}}) \;\middle|\; v \notin D_{t+1}(x^{\mathsf{obj}}) \wedge v > \max\left(D_t(x^{\mathsf{obj}})\right)\right\} \tag{2}$$

$$r_t^{\mathsf{lb}} = \#\left\{v \in D_t(x^{\mathsf{obj}}) \;\middle|\; v \notin D_{t+1}(x^{\mathsf{obj}}) \wedge v < \min\left(D_t(x^{\mathsf{obj}})\right)\right\} \tag{3}$$

$$r_t^{\mathsf{mid}} = \frac{r_t^{\mathsf{ub}} - r_t^{\mathsf{lb}}}{\left|D_1(x^{\mathsf{obj}})\right|} \tag{4}$$

$$r_t^{\mathsf{end}} = -1 \text{ if unfeasible solution found (0 otherwise)} \tag{5}$$

$$r^{\mathsf{acc}} = \left(\sum_{t=1}^{T-1} r_t^{\mathsf{mid}}\right) + r_T^{\mathsf{end}} \tag{6}$$

Figure 3 Intermediate reward when four values are pruned from the domain.

An experimental analysis of this new reward scheme (*propagation-based reward*) is carried out for the *graph coloring*, *maximum cut*, and *maximum independent set* problems; we look at the quality of the solution found after a single dive in the search tree. As a baseline, we consider a reward (*score reward*) that only gives a value at terminal nodes (r_T^{end}) without an intermediate reward. Besides, we also consider the solutions returned by a random value-selection heuristic as a baseline. Figure 4 shows the evolution of the quality of the first solution returned (*y*-axis, averaged on 20 instances of the validation step) with the training time (number of episodes in the *x*-axis) using for training our restart-based search strategy defined in Section 4.1. Instances are Barabási-Albert randomly generated graphs with 50 nodes. Except for the rewarding scheme, the other parts of the architecture are unchanged. We observe that the *propagation-based reward* provides a more stable training (Figure 4a) and can converge to a better model or, at least, to an equally good model as the terminal *score reward* (Figures 4b and 4c).

It should be noted that depending on the problem, the reward signal may remain sparse inside episodes even with our definition; this explains the discrepancy across the three class problems. Indeed, constraint propagation might take several steps to reach the objective variable, meaning that for related intermediate decisions, no value will be pruned from the domain of the objective variable. The *graph coloring* problem is thus the problem for which taking these intermediate rewards is the most beneficial. Indeed, any previously unused color added will negatively impact the domain of the objective function, yielding an insightful negative reward. Conversely, branching on the *maximum independent set* problem does not consistently impact the objective function domain through the mechanism of constraint propagation, particularly at the beginning of the search. Our method yields no worse result than the usual reward signal in this setting. This worst-case scenario empirically validates the robustness of this reward.

(a) Graph coloring. **(b)** Maximum cut. **(c)** Maximum independent set.
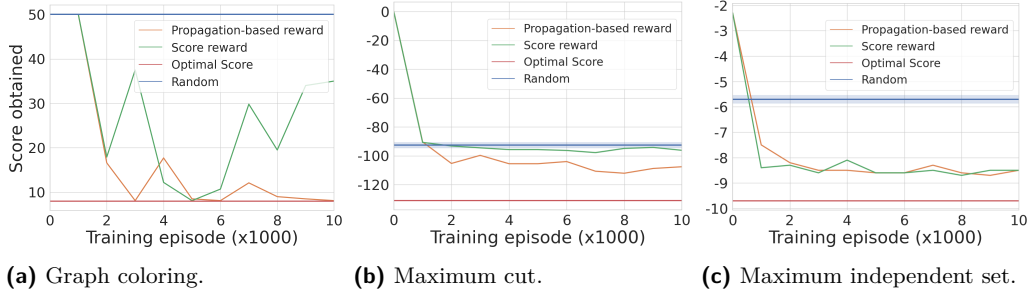
Figure 4 Training curve for the two rewarding schemes, each validation step corresponds to performing a single dive in the search tree, the *score* obtained refers to the quality of the solution found on the leaf node.

4.3 Heterogeneous Graph Neural Network Architecture

An important part of the framework is the neural network architecture that we designed to perform a prediction of the next value to branch on. A high-level representation is proposed in Figure 5. Four steps are carried out: (1) a *CP model encoder*, (2) a *graph neural network encoder*, (3) a *neural network decoder*, and (4) an *action-selection policy*. They are detailed in the next subsections.
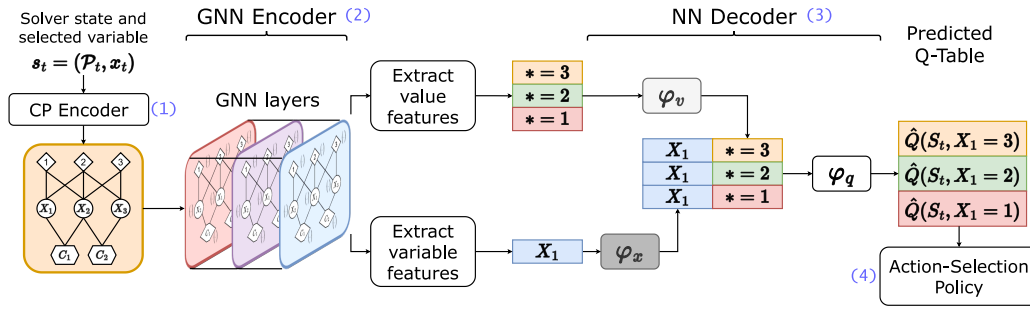

Figure 5 High-level overview of the neural architecture designed.

Step 1: CP Model Encoder

The core idea is to learn for any CP model given as input, unlike Cappart et al. [11], who require a specific encoding for each combinatorial problem. This has been achieved for mixed-integer programs thanks to a bipartite graph representation [16] and by Chalumeau et al. [12] for CP models thanks to a tripartite graph. This last work does not leverage any feature related to the variables, values, or constraints. We built upon this last approach by adding such features. Specifically, let $\mathcal{P} = \langle X, D(X), C, O \rangle$ be the combinatorial problem we want to encode. The idea consists in building a simple undirected graph $\mathcal{G}(V_1, V_2, V_3, f_1, f_2, f_3, E_1, E_2)$ encoding all the information of $\mathcal{P}_t$ from a state $s_t = (\mathcal{P}_t, x_t)$. In this representation, V_1, V_2, and V_3 are three sets of vertices, f_1, f_2, and f_3 are three sets of feature vectors, and E_1 with E_2 are two distinct sets of edges. This yields a graph with three types of nodes decorated with features. The first part of the encoding we propose is as follows: (1) each variable, constraint, and value corresponds to a specific type of node ($V_1 = X$, $V_2 = C$, and $V_3 = D$), (2) each time a variable $x \in V_1$ is involved in a constraint $c \in V_2$, an edge $(x, c) \in E_1$ is added

between both nodes, (3) each time a value $v \in V_3$ is in the domain of a variable $x \in V_1$, an edge $(v, x) \in E_2$ is added between both nodes. This gives a tripartite graph representation of a CP model generically. This is illustrated in Figure 6. The second part of the encoding is to add features to each node. Intuitively, the features will provide meaningful information and thus improve the quality of the model. The features we considered are proposed below. We note that we can easily extend this encoding by integrating new features.

1. *Features attached to variables (f_1)*: the current domain size, the initial domain size, a binary indication if the variable is already assigned, and a binary indication if the variable corresponds to the objective.

2. *Features attached to constraints (f_2)*: the constraint type (one-hot encoding), and a binary indication if the constraint propagation has reduced domains.

3. *Features attached to values (f_3)*: its numerical value.
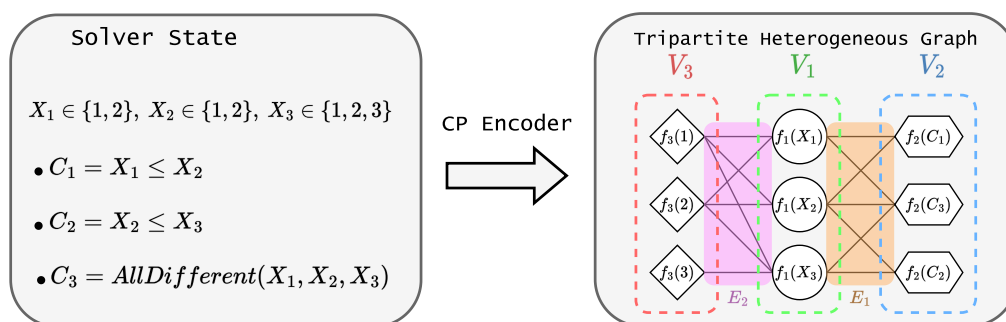
Figure 6 Representation computed by the CP encoder on a simple example.

Step 2: Graph Neural Network Encoder

Once the CP model has been encoded as a graph, the next step is to embed this representation as a latent vector of features for each node of the graph (see Section 3.2). We propose to carry out this operation with a graph neural network. Unlike the standard prediction scheme presented in Equation (1), our graph has three types of nodes. For this reason, we opted for a *heterogeneous* architecture. Concretely, a specific convolution is carried out for each node type. The architecture is detailed in Equations (7) to (9), where $\bigoplus$ is the *sum-pooling* or *mean-pooling* aggregation, operator $(.\|.)$ is a concatenation of vectors, $N_x(n)$ is the set of neighbouring nodes of n from V_1 (variable), $N_c(n)$ is the set of neighbouring nodes of n from V_2 (constraint), $N_v(n)$ is the set of neighbouring nodes of n from V_3 (value), $\theta_{1,\dots,10}^k$ are weight matrices at layer k, and g is the leakyReLU activation function [28]. Another difference with the canonical GNN equation is the integration of *skip connections* (h_x^0, h_c^0, and h_c^0) allowing to keep at each layer information from the input features. This technique is ubiquitous in deep convolutional networks such as in *ResNet* [20]. Finally, the initial embedding are initialized as follows: $h_x^0 = \theta_{11}f_1$, $h_c^0 = \theta_{12}f_2$, and $h_v^0 = \theta_{13}f_3$, where $\theta_{11,\dots,13}$

are new weight matrices.

$$h_x^{k+1} = g\left(\theta_1^k h_x^0 \parallel \theta_2^k h_x^k \parallel \left(\bigoplus_{c \in N_c(x)} \theta_3^k h_c^k\right) \parallel \left(\bigoplus_{v \in N_v(x)} \theta_4^k h_v^k\right)\right) \qquad \forall x \in V_1 \qquad (7)$$

$$h_c^{k+1} = g\left(\theta_5^k h_c^0 \parallel \theta_6^k h_c^k \parallel \left(\bigoplus_{x \in N_x(c)} \theta_7^k h_x^k\right)\right) \qquad \forall c \in V_2 \qquad (8)$$

$$h_v^{k+1} = g\left(\theta_8^k h_v^0 \parallel \theta_9^k h_v^k \parallel \left(\bigoplus_{x \in N_x(v)} \theta_{10}^k h_x^k\right)\right) \qquad \forall v \in V_3 \qquad (9)$$

Step 3: Neural Network Decoder

At this step, a d-dimensional tensor is obtained for each graph node. Let $x \in V_1$ be the node representing the current variable selected for branching, and $V_x \subseteq V_3$ the subset of nodes representing the values available for x (i.e., the values that are in the domain of the variable). The goal of the *decoder* is to predict a *Q-value* (see Section 3.1) for each $v \in V_x$. The computation is formalized in Equation (10), where h_x^K and h_v^K are the node embedding of variable x and value v, respectively, after K iterations of the GNN architecture. The functions $\varphi_x : \mathbb{R}^d \rightarrow \mathbb{R}^l, \varphi_v : \mathbb{R}^d \rightarrow \mathbb{R}^l, \varphi_q : \mathbb{R}^{2l} \rightarrow \mathbb{R}$ are fully-connected neural networks. Such a Q-value must be computed for each value $v \in V_x$. It is internally done thanks to matrix operations, allowing a more efficient computation.

$$\hat{Q}(h_x^K, h_v^K) = \varphi_q\left(\varphi_x(h_x^K) \parallel \varphi_v(h_v^K)\right) \quad \forall v \in V_x \qquad (10)$$

Step 4: Action-Selection Policy

Once all the Q-values have been computed for the current variable, the policy is defined by an *explorer* that can decide to exploit the approximated Q-values by greedily choosing the best action as shown in Equation (11) or decide to select unpromising action associated with a lower Q-value (for example, by selecting a random action with probability ϵ). This behavior derives from the trade-off between exploitation and exploration, which is necessary for early learning when the estimates of Q-values are poor, and when only a few states have been visited. Once trained, the Q-values should represent the branching choice leading to the best decision according to the reward of Equation (6).

$$\pi(v|x) = \mathsf{argmax}_{v \in V_x} \hat{Q}(h_x^K, h_v^K) \qquad (11)$$

Assembling all the pieces, this architecture gives a generic approach to obtaining a data-driven value-selection heuristic inside a CP solver. Concerning the search strategy used for evaluation (which is different from the *restart-based* one used for training), we propose to embed our predictions inside an *iterative limited discrepancy search* (ILDS) [19]. This strategy is commonly used when we are confident in the quality of the heuristic. The core idea is to restrict the number of branching choices deviating from the heuristic (i.e., a *discrepancy*). By doing so, the search will explore a subset of solutions expected to be good while giving a chance to reconsider the value-heuristic selection which is nevertheless prone to errors. This mechanism is enriched with a procedure that iteratively increases the number of discrepancies allowed once a level has been explored.

5 Experiments

The goal of this section is to evaluate the quality of the learned value-selection heuristic and the efficiency of the approach. Three combinatorial optimization problems are considered: *graph coloring* (COL), *maximum independent set* (MIS), and *maximum cut* (MAXCUT).

5.1 Experimental Protocol

Three configurations for the distribution of the problems generated are proposed for each problem: *small* (20 to 30 nodes), *medium* (40 to 50 nodes), and *large* (80 to 100 nodes) instances, except for MAXCUT which was already challenging for the *medium* size. Training is carried out on randomly generated Barabási-Albert graph [2] with a density factor varying between 4 and 15 according to the size of the instances. A specific model is trained for each configuration of each combinatorial problem. The training is done using randomly generated instances. Evaluation is then performed on 20 new graphs following the same distributions. The models are trained on an Nvidia Tesla V100 32Go GPU until convergence. It took up to 72 hours of training time for the most difficult cases (*graph coloring* with 80 nodes) and less than 1 hour for the simplest cases (*graph coloring* with 20 nodes). Each operation of the CP solver during training and evaluation is carried out on a CPU Intel Xeon Silver 4116 at 2.10GHz. The approach has been implemented in *Julia* and is integrated into the solver *Seapearl*. The implementation is available on GitHub with BSD 3-Clause licence[2].

We compared our approach (Learned, ILDS) with two other generic value selection heuristics: *impact-based search* (Impact) [37] and *activity-based search* (Activity) [31]. The standard *minDomain* heuristic is used for the variable selection. Comparisons with Chalumeau et al. [12] have been provided in Section 4.1. As it has been highlighted that this approach is not suited to find good solutions quickly, it is not included again in the next experiments. Each approach is evaluated with a fixed node budget depending on the parameters of the distribution used to generate the problems. For our approach, the performance obtained after the first dive in the tree search is also monitored (Learned, 1st dive). As Impact and Activity are online learning methods, they perform similarly to a random selection at the beginning of the search. For this reason, the performance obtained after the first dive in the tree search with such methods is omitted. Finally, we also included a comparison with a random selection using DFS with the same node budget (Random). Finally, the optimal cost (OPT) has been obtained with an exact approach without any restriction on the budget.

5.2 Quantitative Results

Table 1 summarizes the main results of our approach. As a general comment, our approach can find solutions of superior quality given a node budget or find the optimal solution by exploring fewer nodes than the baselines. Interestingly, our approach (Learned, ILDS) can learn a branching strategy giving high-quality solutions, even without backtracking (1st dive). For instance, a single dive for *maximum cut* with 50 nodes yields almost instantly a solution with an optimality gap of 0.16, whereas a depth-first search with a random selection (Random, DFS) required 19 seconds and roughly 53,000 nodes explored to find a solution with the same gap. Within this same budget, (Learned, ILDS) significantly improves the solution and achieves an optimality gap of 0.09. It is worth highlighting that (Learned, ILDS) took 130 seconds to explore 38,744 nodes and has, thereby, an exploration rate slower than the other methods. This significantly increased execution time is mainly because calling the graph neural network architecture (Section 4.3) at each tree search node is much more computationally expensive than calling a simple heuristic. This difficulty is further discussed in Section 5.3.

Concerning Activity and Impact heuristics, they yield no improvement on *graph coloring* compared to a random strategy. This can be explained by the fact that this class of problem has many possible combinations of variables and values for branching. This requires a


■ **Table 1** Results for the three problems given a fixed node budget. The average result (rounded) on the 20 test instances is reported for each configuration. *Gap* indicates the optimality gap, *Node* gives the number of nodes explored before finding the best solution within the budget, and *Time* gives the time (seconds) before finding this solution.

| | | | Learned | | | | Activity-Based | | | Impact-Based | | | Random | | | |
| | | 1st dive | ILDS | | | DFS | | | DFS | | | DFS | | | |
	Size	OPT	Gap	Gap	Node	Time	Gap	Node	Time	Gap	Node	Time	Gap	Node	Time	Budget
COL	20	5.05	0.06	0	27	< 1	0	378	< 1	0	374	< 1	0	378	< 1	10^3
	40	7.90	0.08	0	104	< 1	0	1,664	< 1	0	1732	< 1	0	1735	< 1	10^4
	80	8.75	0.06	0	120	1	0	7,051	2	0	7,057	2	0	7,211	2	10^5
MIS	30	9.90	0.08	0	88	< 1	0	215	< 1	0	297	< 1	0	293	< 1	10^3
	50	15.00	0.09	0	539	1	0	5,807	1	0	7,474	1	0	8,942	1	10^4
	100	21.70	0.20	0.02	28,392	253	0.09	35,536	7	0.10	38,154	8	0.10	41,774	9	10^5
MAXCUT	20	46.70	0.15	0.03	3,714	5	0.04	4,635	1	0.03	5,959	2	0.04	4877	1	10^4
	50	222.00	0.16	0.09	38,744	130	0.17	44,664	14	0.17	47,970	17	0.17	53,110	19	10^5

significantly larger number of explored nodes to initialize these two heuristics efficiently. For the two other problems, characterized by a binary domain for the values to branch on, Activity and Impact provide significantly better results than the random strategy, which is the expected behavior. In all the tested situations, (Learned, ILDS) provides the best optimality gap within the node budget. Additional results are proposed in Figure 7 using performance profiles [14] for the two hardest situations (100 for *maximum independent set*, and 50 for *maximum cut*) given a node budget of 100 or 1000 nodes.

(a) Maximum cut with 50 nodes.

(b) Maximum independent set with 100 nodes.
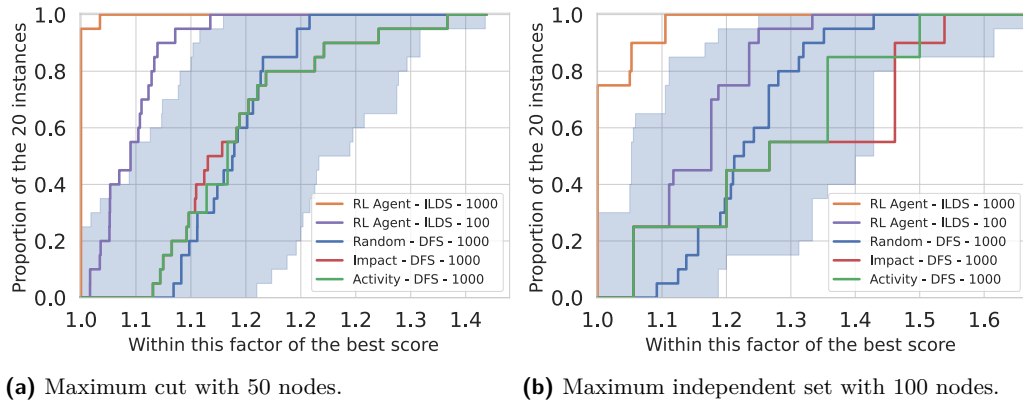
■ **Figure 7** Best solutions found within a restricted node budget on largest instances for the three problems considered. We set a small budget to evaluate the ability of each approach to *find quickly a good solution*, which is the objective aimed by this work. The performance profile ratio is computed using the optimal solution as a reference. Within the same maximal number of nodes visited (1000), we observe that (Learned, ILDS) dominate all the other methods. Besides, we still perform better than the baselines when restricting ten times the budget for ILDS-Learned.

5.3   Discussions and Opportunities of Further Research

The previous experiments showcased the promise of this framework to quickly find good solutions towards a generic value-selection heuristic inside a CP solver. There are nonetheless open challenges that must be considered for practical use. Four of them are discussed.

Challenge 1: Scalability of the Representation

Our approach faces a double penalty regarding its *scaling* capability: as the problem grows larger, the tripartite representation increases significantly in size, which results in a longer computation time required to make one branching decision. This impacts both training and evaluation. Additionally, the number of nodes (and, therefore, decisions to be made) in the search tree grows exponentially with the problem size, exacerbating the aforementioned phenomenon. Consequently, our approach is penalized twice due to the exponential behavior of combinatorial problems. As a concrete example, *graph coloring* instances with 80 nodes require 72 hours of training on a GPU, while only 1 hour is required for the smallest instances. An interesting research direction to mitigate this difficulty is to build a mechanism to compact the representation, for instance, thanks to network pruning tools [51] or with *transfer learning*. Another idea is to call the model only in a few nodes, in a similar fashion as Cappart et al. [9] did for *decision-diagram-based branch-and-bound* [6]. On a lower level of computation, standard constraint programming solvers perform sequential decisions and are therefore optimized for CPU architecture. Concerning the training, it is carried out on a GPU. In the current implementation, each branching decision requires loading the entire tripartite graph on the Video RAM, which is inefficient. We believe much work could be done to optimize this CPU/GPU architecture, for instance by delegating other operations on the GPUs, such as the propagation of few constraints [8, 46].

Challenge 2: Tackling Highly Constrained Problems

The experiments proposed in the paper considered combinatorial problems where the difficulty lay in finding the *best* solution. Still, it was easy to find a *feasible* solution, even of poor quality. We empirically observed that the learning performance largely depends on the abundance of feasible solutions in the search space. This is explained by the definition of the reward, which is based on the propagation occurring on the objective variable (see r_t^{mid} in Section 4.2). However, when feasible solutions are not easily obtained, such as in highly constrained problems, the reward signal becomes less informative. Addressing such combinatorial problems remains an open challenge. We believe an extension of the reward signal can address this in order to handle other situations.

Challenge 3: Learning a Combined Variable/Value Heuristic

Although this work proposes to learn a value-selection heuristic, learning how to branch on variables has already been considered in the literature [42]. An interesting research direction is to adapt this architecture to learn a variable-selection and a value-selection heuristic in a unified way. A possible direction is to consider a model with a double-head decoder, the first for selecting the variable and the second for selecting the value. On the training aspect, two reinforcement learning agents could be trained, with an the incentive to cooperate with the information sharing [44].

Challenge 4: Proving the Optimality of a Solution

The goal pursued in this paper is to find the best solution as quickly as possible. Another direction is to guide the search to speed-up the optimality proof. It is what has been proposed by Chalumeau et al. [12]. In practice, finding good solutions and proving optimality are complementary aspects inside a constraint programming solver and should be both considered. Possible directions to do so could be to redefine the reward function appropriately or to revise the definition of an episode, as proposed by Scavuzzo et al. with TreeMDPs [38].

6   Conclusion

The efficiency of constraint programming solvers is partially due to the branching heuristics used to guide the search. In practice, value-selection heuristics are often designed thanks to problem-specific expert knowledge, often out of reach for non-practitioners. In this paper, we proposed a method based on reinforcement learning for obtaining such a heuristic, thanks to historical data, characterized by problem instances following the same distribution of the one that must be solved. This has been achieved thanks to a restart-based training procedure, a non-sparse reward signal, and a heterogeneous graph neural network architecture. Experiments on three combinatorial optimization problems show that the framework can find better solutions close to optimality in fewer nodes visited than other generic baselines. Several limitations and challenges (e.g., tractability for larger or real-world instances, transfer learning, sparsity of the reward signal) have been identified, and addressing them is part of future work. We also plan to consider other combinatorial problems, such as the ones proposed in XCSP3 competitions [3].

References

1 Joshua Achiam. Spinning up as a deep RL researcher, October 2018. URL: `spinningup.openai.com/en/latest/spinningup/spinningup.html`.

2 Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

3 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2022 XCSP3 competition. *arXiv preprint arXiv:2209.00917*, 2022.

4 Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

5 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

6 David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

7 Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604. Springer, 2018.

8 Federico Campeotto, Alessandro Dal Palu, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of GPUs in constraint solving. In *Practical Aspects of Declarative Languages: 16th International Symposium, PADL 2014, San Diego, CA, USA, January 20-21, 2014. Proceedings 16*, pages 152–167. Springer, 2014.

9 Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 2022.

10 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. URL: `http://jmlr.org/papers/v24/21-0449.html`.

11 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*, volume 35, pages 3677–3687, 2021.

12 Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. In *International Conference*

on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pages 392–409. Springer, 2021.

13 Geoffrey Chu and Peter J. Stuckey. Learning value heuristics for constraint programming. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimization Problems 2015*, pages 108–123. Springer, 2015.

14 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

15 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.

16 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.

17 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.

18 Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

19 William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th international joint conference on Artificial intelligence*, volume 1, pages 607–613, 1995.

20 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

21 Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer, 2011.

22 Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

23 Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

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

25 Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL: `https://openreview.net/forum?id=ByxBFsRqYm`.

26 Markus Kruber, Marco E Lübbecke, and Axel Parmentier. Learning when to use a decomposition. In *International conference on AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 202–210. Springer, 2017.

27 Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

28 Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, Georgia, USA, 2013.

29 Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

30 Laurent Michel, Pierre Schaus, and Pascal van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13:133–184, 2021.

31   Laurent Michel and Pascal van Hentenryck. Activity-based search for black-box constraint programming solvers. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 228–243. Springer, 2012.

32   Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961. `doi:10.1109/JRPROC.1961.287775`.

33   Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

34   Mouad Morabit, Guy Desaulniers, and Andrea Lodi. Machine-learning–based column selection for column generation. *Transportation Science*, 55(4):815–831, 2021.

35   Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

36   Jean-Yves Potvin, Danny Dubé, and Christian Robillard. A hybrid approach to vehicle routing using neural networks and genetic algorithms. *Applied Intelligence*, 6(3):241–252, 1996.

37   Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571. Springer, 2004.

38   Lara Scavuzzo, Feng Yang Chen, Didier Chételat, Maxime Gasse, Andrea Lodi, Neil Yorke-Smith, and Karen Aardal. Learning to branch with tree MDPs. *arXiv preprint arXiv:2205.11107*, 2022.

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

40   John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

41   Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 336–353. Springer, 2019.

42   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.

43   Haoran Sun, Wenbo Chen, Hui Li, and Le Song. Improving learning to branch via reinforcement learning. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

44   Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning, 2017. `arXiv:1706.05296`.

45   Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

46   Fabio Tardivo and Agostino Dovier. Constraints propagation on GPU: A case study for alldifferent. In *Proceedings of the 37th Italian Conference on Computational Logic*, 2022.

47   Alexander Trott, Stephan Zheng, Caiming Xiong, and Richard Socher. Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

48   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.

49   Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

50   Wei Xia and Roland Yap. Learning robust search strategies using a bandit-based approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

51   Xin Yu, Thiago Serra, Srikumar Ramalingam, and Shandian Zhe. The combinatorial brain surgeon: Pruning weights that cancel one another in neural networks. In *International Conference on Machine Learning*, pages 25668–25683. PMLR, 2022.