# **Singapore Management University**

# Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems School of Computing and Information Systems

3-2022

# Learning variable ordering heuristics for solving constraint satisfaction problems

Wen SONG

Zhiguang CAO Singapore Management University, zgcao@smu.edu.sg

Jie ZHANG

Chi XU

Andrew LIM

Follow this and additional works at: https://ink.library.smu.edu.sg/sis\_research

Part of the Artificial Intelligence and Robotics Commons

# Citation

SONG, Wen; CAO, Zhiguang; ZHANG, Jie; XU, Chi; and LIM, Andrew. Learning variable ordering heuristics for solving constraint satisfaction problems. (2022). *Engineering Applications of Artificial Intelligence*. 109, 1-12.

Available at: https://ink.library.smu.edu.sg/sis\_research/8070

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg. Contents lists available at ScienceDirect



Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

# Learning variable ordering heuristics for solving Constraint Satisfaction Problems



Engineering Applications Artificial Intelligence

Wen Song<sup>a</sup>, Zhiguang Cao<sup>b,\*</sup>, Jie Zhang<sup>c</sup>, Chi Xu<sup>b,c</sup>, Andrew Lim<sup>d</sup>

<sup>a</sup> Institute of Marine Science and Technology, Shandong University, China

<sup>b</sup> Singapore Institute of Manufacturing Technology (SIMTech), Singapore

<sup>c</sup> School of Computer Science and Engineering, Nanyang Technological University, Singapore

<sup>d</sup> School of Computing and Artificial Intelligence, Southwest Jiaotong University, China

#### ARTICLE INFO

Keywords: Constraint Satisfaction Problem Variable ordering Deep reinforcement learning Graph Neural Network

# ABSTRACT

Backtracking search algorithms are often used to solve the Constraint Satisfaction Problem (CSP), which is widely applied in various domains such as automated planning and scheduling. The efficiency of backtracking search depends greatly on the variable ordering heuristics. Currently, the most commonly used heuristics are hand-crafted based on expert knowledge. In this paper, we propose a deep reinforcement learning based approach to automatically discover new variable ordering heuristics that are better adapted for a given class of CSP instances, without the need of relying on hand-crafted features and heuristics. We show that directly optimizing the search tree size is not convenient for learning, and propose to optimize the expected cost of reaching a leaf node in the search tree. To capture the complex relations among the variables and constraints, we design a representation scheme based on Graph Neural Network that can process CSP instances with different sizes and constraint arities. Experimental results on random CSP instances show that on small and medium sized instances, the learned policies outperform classical hand-crafted heuristics with smaller search tree (up to 10.36% reduction). Moreover, without further training, our policies directly generalize to instances of larger sizes and much harder to solve than those in training, with even larger reduction in the search tree size (up to 18.74%).

#### 1. Introduction

Combinatorial problems widely exist in many domains (Petit and Trapp, 2019). As one of the most commonly studied combinatorial problems in computer science and artificial intelligence, Constraint Satisfaction Problem (CSP) provides a general framework for modeling and solving combinatorial problems. Solving CSP plays a central role in many research areas. A typical and successful application of CSP is automated planning and scheduling (Salido et al., 2008), which has numerous real-world applications, ranging from industry 4.0 (Legat and Vogel-Heuser, 2017; Song et al., 2019) to robotics (Kasprzak et al., 2014; Behrens et al., 2019). A CSP instance involves a set of variables and constraints. To solve it, one needs to find a value assignment (i.e. solution) for all the variables such that all the constraints are satisfied, or prove such assignment does not exist. Despite its ubiquitous applications, unfortunately, CSP is well known to be NP-complete in general (Mackworth and Freuder, 1993). To solve CSP efficiently, backtracking search algorithms are often employed, which are exact algorithms with the guarantee that a solution will be found if one exists. Though the worst-case complexity is still exponential, with the help of constraint propagation, backtracking search algorithms often perform reasonably well in practice (Rossi et al., 2006).

In general, a backtracking search algorithm performs depth-first traverse of a search tree, and tries to find a solution by iteratively selecting a variable and applying certain branching strategy. The decision of which variable to select next is referred to as variable ordering. It is well acknowledged that the choice of variable ordering has a critical impact on the efficiency of backtracking search algorithms (Gent et al., 1996). However, finding the optimal ordering, i.e. the one results in a minimalsized search tree, is at least as hard as solving the CSP (Liberatore, 2000; Rossi et al., 2006). Therefore, current practice mainly relies on hand-crafted variable ordering heuristics obtained from the experience of human experts, such as MinDom (Haralick and Elliott, 1980), Dom/Ddeg (Bessiere and Régin, 1996), and impact-based heuristic (Refalo, 2004). Though they are easy to use and widely adopted, they do not have any formal guarantees on the optimality. In addition, they are designed for solving any CSP instance without considering the problemspecific structures, which can be exploited to achieve much better efficiency. However, incorporating these additional features requires

\* Corresponding author.

*E-mail addresses:* wensong@email.sdu.edu.cn (W. Song), zhiguangcao@outlook.com (Z. Cao), zhangj@ntu.edu.sg (J. Zhang), cxu@simtech.a-star.edu.sg (C. Xu), i@limandrew.org (A. Lim).

https://doi.org/10.1016/j.engappai.2021.104603

Received 3 June 2021; Received in revised form 5 November 2021; Accepted 5 December 2021 Available online 20 December 2021 0952-1976/© 2021 Elsevier Ltd. All rights reserved. substantial experience and deep domain knowledge, which are hard to obtain in reality (Rossi et al., 2006).

Recently, Deep Neural Networks (DNNs) have been shown to be promising in learning algorithms for solving NP-hard problems, such as routing, graph problems, Propositional Satisfiability Problem (SAT), and so on Bengio et al. (2020). The effectiveness comes from the fact that given a class of problem instances (e.g. drawn from a distribution), DNN can be trained to discover useful patterns that may not be known or hard to be specified by human experts, through supervised or reinforcement learning (RL) (Ma et al., 2020; Altan et al., 2021). Based on the above success, in this paper, we ask the following question: can we use DNN to discover better variable ordering heuristics for a class of CSP? This is not a trivial task, due to the following challenges. Firstly, given the exponential (worst-case) complexity of CSP, it is not practical to obtain large amount of labeled training data (e.g. optimal search paths), therefore it is hard to apply supervised learning methods. Secondly, CSP instances have different sizes in terms of number of variables and constraints, and the constraint arities (i.e. number of variables involved) are also different. It is crucial to design a deep representation scheme that can effectively process CSP instances of any size and constraint arity.

In the literature, several works have tried to leverage machine learning techniques to learn variable ordering heuristics for solving satisfaction problems, such as SAT (Lagoudakis and Littman, 2001), Quantified Boolean Formulas (QBF) (Samulowitz and Memisevic, 2007), and CSP (Epstein and Petrovic, 2007; Xu et al., 2009). However, as will be detailed in Section 2, these methods cannot directly generate ordering policies from solving status. They require a set of predefined heuristics from which the algorithm learns to choose (or combine), and are all based on conventional learning methods and hand-crafted features. More recently, independent to our work, DNN is used with RL to learn value ordering heuristics for CSP (Cappart et al., 2021; Chalumeau et al., 2021). However, as mentioned in Chalumeau et al. (2021), learning variable ordering heuristics with DNN raises additional challenges to the learning mechanism.

In this paper, we design a deep reinforcement learning (DRL) agent which tries to make the optimal variable ordering decisions at each decision point to minimize the search tree size. More specifically, variable ordering in backtracking search is modeled as a Markov Decision Process (MDP), where the optimal policy is to select at each decision point the variable with the minimum expected number of search nodes. The DRL agent can optimize its policy by learning from its own experiences of solving CSP instances drawn from a distribution, without the need of supervision. However, as will be shown later, such direct formulation could cause inefficiency and inconvenience to the learning mechanism, since learning must be delayed until backtracking from a search node. To resolve this issue, we consider the search paths originated from a node as separate trajectories, and opt to minimize the expected number of remaining nodes to reach a leaf node. We represent the internal states of the search process based on Graph Neural Network (GNN) (Xu et al., 2019), which can process CSP instances of any size and constraint arity, and effectively capture the relationship between the variables and constraints. We use Double Deep Q-Network (DDQN) (Hasselt et al., 2016) to train the GNN based RL agent. Experimental results on random CSP instances generated by the well-known model RB (Xu et al., 2007) show that the RL agent can discover policies that are better than the traditional hand-crafted variable ordering heuristics, in terms of minimizing the search tree size. More importantly, the learned policy can effectively generalize to larger instances that have never been seen during training.

To summarize, the contributions of this paper are as follows:

• We propose a novel DRL method to learn variable ordering heuristics for solving a class of CSP. Different from previous works, our method learns in an end-to-end fashion, meaning that through self-training, it can automatically generate high-quality ordering heuristics from raw state input without the need of hand-crafted features and predefined heuristics.

- We propose to minimize the expected search effort of reaching the leaf nodes, instead of directly minimizing the search tree size. In this way, RL training could be more efficient since temporal difference learning is effectively enabled. Specifically, the agent can learn at each search step, instead of waiting until backtracking.
- We propose a GNN based scheme to represent the internal search states, based on which a deep Q network is designed to output the Q-value of each candidate variable end-to-end from raw state features of variables and constraints. Such representation is agnostic to the instance size, hence enables generalizing to large unseen instances.

The rest of this paper is organized as follows. We first summarize related works in Section 2, followed by preliminaries about CSP in Section 3. Then in Section 4, we present our deep RL method in detail. Section 5 provides the experimental results and analysis. Section 6 further discusses the reasons of the good performance of our method, as well as the limitations. Finally, Section 7 concludes the paper.

#### 2. Related work

Recently, there has been an increasing attention on using deep (reinforcement) learning to tackle hard combinatorial (optimization or satisfaction) problems. Quite a few works in this direction focus on solving specific types of problems, including routing (Kool et al., 2019; Wu et al., 2021; Xin et al., 2021b,a), graph problems (Khalil et al., 2017a; Li et al., 2018), and scheduling (Zhang et al., 2020; Mao et al., 2019). Instead of solving specific problems, we focus on CSP which is a general representation of combinatorial problems.

In the literature, a number of methods try to tackle satisfaction problems such as CSP and SAT in an end-to-end fashion, meaning that training DNN to directly output a solution for a given instance. Xu et al. (2018) represent binary CSP as a matrix and train a Convolutional Neural Network (CNN) to predict its satisfiability, but cannot give the solution for satisfiable instances. In addition, the matrix representation scheme cannot scale to arbitrary problem size. Galassi et al. (2018) train a DNN that can construct a feasible solution of a CSP instance by extending a partial assignment, however the representation scheme based on one-hot encoding of assignment also suffers from the issue of being restricted to a pre-determined problem size. Selsam et al. (2019) train a satisfiability classifier for SAT, which can be considered as a special case of CSP. The underlying architecture is based on GNN instead of CNN, therefore can process instances with different sizes. The authors also provide an unsupervised procedure to decode a satisfying assignment. Amizadeh et al. (2019) propose a differentiable architecture to train a GNN that directly aiming at solving the Circuit-SAT problem instead of only predicting its satisfiability.

Despite their simplicity and effectiveness, as pointed out by Bengio et al. in a recent survey (Bengio et al., 2020), end-to-end methods suffer from two major limitations: (1) feasibility is weak since it is hard for them to handle advanced types of constraints, and (2) no guarantee on the solution quality (in terms of optimality and feasibility for optimization and satisfaction problems, respectively). A more promising way is to apply machine learning within the framework of exact algorithms, such that the feasibility and solution quality can be guaranteed (Bengio et al., 2020). A typical exact framework is the branch-and-bound algorithm for solving Mixed Integer Linear Programs (MILPs). He et al. (2014) use imitation learning to learn a control policy for selecting branches in the branch-and-bound process. Khalil et al. (2016) achieves similar purpose by solving a learning-to-rank task to mimic the behaviors of strong branching. Khalil et al. (2017b) also develop a machine learning model to decide whether the primal heuristics should be run for a given branch-and-bound node. These methods are based on linear models with static and dynamic features describing the current branch-and-bound status. More recently, Gasse et al. (2019) use imitation learning to mimic strong branching, where the underlying states are represented using GNN. Similarly, a GNN

based network is designed in Ding et al. (2020), which is trained in a supervised way to predict values of binary variables in MILP. Though sharing similar GNN structure, our work differs from Gasse et al. (2019) and Ding et al. (2020) in mainly two aspects. First, our method does not require labels that are costly to obtain but necessary for imitation or supervised learning. Second, as will be shown latter, we only uses 4 simple raw features, while Gasse et al. (2019) and Ding et al. (2020) rely on 19 and 22 complex MILP features, respectively.

Another exact framework is the backtracking search algorithms for solving satisfaction problems. Balafrej et al. (2015) use bandit model to learn a policy that can adaptively select the right constraint propagation levels at each node of a CSP search tree. More close to our work, several methods use traditional machine learning to choose the branching heuristics for solving CSP and some special cases. Lagoudakis and Littman (2001) use RL to learn the branching rule selection policy for the #DPLL algorithm for solving SAT, which requires finding all solutions for a satisfiable instance. However, as will be discussed in Section 4, this RL formulation is not directly applicable for learning in our case. Samulowitz and Memisevic (2007) study the heuristic selection task for solving Quantified Boolean Formulas (QBF), a generalization of SAT, through supervised learning. In terms of CSP, Epstein and Petrovic (2007) opt to learn a linearly weighted profile of multiple ordering heuristics to select the next variable and value for each search node. Though their training mechanism is self-supervised by using the solver's own solving experiences, it is not formulated as a RL task and the weight of each heuristic is learned simply based on the frequency it supports correct or oppose incorrect decisions. Xu et al. (2009) propose a RL formulation for variable ordering heuristic selection, but only provide preliminary results.

Though sharing similar goals, our approach significantly differs from Lagoudakis and Littman (2001), Samulowitz and Memisevic (2007), Epstein and Petrovic (2007) and Xu et al. (2009) in several ways. Firstly, we propose a RL formulation that are suitable for temporal difference learning during backtracking search, instead of wait until solving is complete. Secondly, in our approach, the learned policy directly picks the next variable based on its own estimates of the environment, without the need of consulting a set of predefined heuristics. Finally, our approach can leverage the approximation and expressive power of DNN. Our GNN based representation scheme provides an effective way to capture the complex relations among variables and constraints of CSP. More importantly, it can effectively process instances of arbitrary sizes and constraint arities which is not viable for the existing deep representations of CSP in Xu et al. (2018) and Galassi et al. (2018).

Concurrent and independent to our work, Cappart et al. (2021) formulate the CSP search process as dynamic programming, and employ deep RL to learn value ordering heuristics, which choose a value for the selected variable to initiate. This work is further extended in Chalumeau et al. (2021), which fully embeds deep reinforcement learning within the CSP solver. Different from these two works, we focus on learning variable ordering policies which is critical to the search performance and could possess unique requirements on the learning mechanism. In the literature, research on learning variable ordering heuristics based on deep RL is rather sparse (Popescu et al., 2021).

#### 3. Preliminaries

A Constraint Satisfaction Problem (CSP) can be formally defined on a *constraint network* (Balafrej et al., 2015), which is a triple  $P = \langle \mathcal{X}, D, C \rangle$ , where  $\mathcal{X} = \{x_1, \dots, x_n\}$  is a set of *n* variables indexed by *i*,  $D = \{d(x_1), \dots, d(x_n)\}$  is the domain of each  $x_i$ , and  $C = \{c_1, \dots, c_e\}$  is a set of *e* constraints indexed by *j*. A domain  $d(x_i)$  is a finite set of values that can be assigned to  $x_i$ . A constraint  $c_j$  is a pair  $c_j = \langle scp(c_j), rel(c_j) \rangle$ , where  $scp(c_j) \subseteq \mathcal{X}$  is the scope of  $c_j$  specifying the variables involved in  $c_j$ , and  $rel(c_j)$  is the relation containing all the allowed value combinations (tuples) of variables in  $scp(c_j)$ . The cardinality of  $scp(c_j)$ , i.e. the number of variables involved in  $c_j$ , is called the arity of the constraint. In this paper, we assume *C* contains only table constraints, i.e. all the allowed tuples for a constraint are explicitly listed as a table. This is somewhat limited, but table constraints are one of the most fundamental and useful constraint types since they can theoretically represent any constraint of other type (Demeulenaere et al., 2016). A solution to the constraint network is an assignment of all the variables such that all the constraint network,<sup>1</sup> or prove no solution exists, i.e. the CSP is unsatisfiable.

The backtracking search process can be considered as performing a depth-first traverse of the search tree, which is dynamically constructed during the search process. At each node, the algorithm selects a variable from those have not been assigned a value yet (i.e. unbounded) according to some variable ordering heuristic, and then, based on certain branching strategy, generates multiple child nodes by posting a set of mutually exclusive and exhaustive branching constraints and performing constraint propagation (CP). Essentially, CP is used to remove some values that are not consistent with the current branching decisions, which can significantly reduce the search space and is the key to achieve high computational efficiency. Hence, each search node corresponds to a subinstance of the original CSP instance being solved, with the same constraints (ignoring branching constraints) and smaller domains. If the domain of some variable is empty after constraint propagation, then the corresponding node is a dead-end since the current branching decisions cannot lead to any feasible solution, and the algorithm backtracks. Search terminates when a solution is found, or the search tree has been completely traversed, meaning that the instance is unsatisfiable. Therefore, the leaf nodes of a search tree include dead-ends and the one with the feasible solution, if one exists.

For backtracking search, one of the most commonly used branching strategies is enumeration, or *d*-way branching, where each child node corresponds to instantiating the selected variable x with a value l in its domain (i.e. posting x = l). The selection of *l* is based on certain value ordering heuristic. Unlike enumeration, the binary branching strategy, or 2-way branching, generates two children for a search node, where a value *l* of the selected variable is applied (i.e. posting x = l) on the left branch and refuted (i.e. posting  $x \neq l$ ) on the right branch. Another popular alternative is domain splitting, which posts two branching constraints based on the selected variable x and value l, e.g.  $x \le l$  and x > l, to produce two children. A simple illustration of these branching strategies is shown in Fig. 1. Our approach is not limited to a particular strategy, since it assumes the general search tree structure. Note that a search node is created only when it is visited by the algorithm. For example, suppose the underlying CSP instance in Fig. 1(a) is satisfiable. When reaching the parent node  $s^E$ , child  $s_1^E$  is visited first. If the algorithm finds out  $s_1^E$  is infeasible, it backtracks to the parent and then creates  $s_2^E$ . If a solution is found under  $s_2^E$ , the algorithm terminates therefore  $s_3^E$  and  $s_4^E$  will not be visited.

# 4. Method

In this section, we formally describe our proposed approach. We first formulate the variable ordering heuristic discovery as a reinforcement learning task. Then, we present our GNN based state representation scheme. Finally, we describe the training algorithm in detail.

<sup>&</sup>lt;sup>1</sup> Generally, one may require to find more than one, or even all solutions, if the CSP instance is satisfiable. While we assume finding one is enough, our approach can be applied when more solutions are required.



**Fig. 1.** An illustration of three branching strategies. The blue and orange circles are parent and child nodes, respectively, and the green boxes are branching constraints. In this example, a variable x with domain  $d(x) = \{1, 2, 3, 4\}$  is selected. The four values are ordered ascendingly for branching. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

#### 4.1. The reinforcement learning formulation

Our goal in this paper is to train a reinforcement learning (RL) agent to perform variable ordering with the objective of minimizing the search tree size, defined as the total number of search nodes. To formulate the RL task, we first need to define the underlying Markov Decision Process (MDP), where the agent is responsible for making the variable ordering decisions, and the solver is considered as the environment. Here we define a state *s* as the instance (for the root node) or subinstance associated with a search node. The states for the leaf nodes are defined as terminal states. Given state *s*, an action *a* is to select an unbounded variable for branching, hence we define the action set as  $A(s) = \{x_i \in \mathcal{X} \mid |d(x_i)| > 1\}$ . Given a simple transition (*s*, *a*, *s'*), we define the cost r(s, a, s') = 1, meaning that one more search node is visited.

However, the actual state transition in backtracking search is not the simple ones. Since a search node could have multiple child nodes, state transitions are not one-to-one as in typical MDP, but one-to-many. Consider the example in Fig. 1(a), the state on the parent  $s^E$  will transit to two subinstances  $s_1^E$  and  $s_2^E$  when action x is taken. Nevertheless, this is not a serious issue because the following transitions from the child nodes are *independent* with each other. In other words, this can be considered as "cloning" the same MDP multiple times, which continue their own transitions thereafter. Based on this observation, for a state s and action  $a \in A(s)$ , let S'(s, a) be the set of next states. Then the reward of taking *a* in *s* is  $r(s, a) = \sum_{s' \in S'(s,a)} r(s, a, s') = |S'(s, a)|$ . Therefore, for a deterministic policy  $\pi$ , the value  $v^{\pi}(s)$  of a state *s* corresponds to the number of search nodes needed to solve the subinstance s following  $\pi$ , if the discounting factor  $\gamma = 1$ . The goal of the RL agent is to find the optimal policy  $\pi^*$  such that the expected (discounted) search tree size is minimized. The optimal action-value function  $Q^*(s, a)$  can be expressed recursively based on the following Bellman optimality equation<sup>2</sup>:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{S'} Pr(S'|s,a) \sum_{s' \in S'} \min_{a' \in A(s')} Q^*(s',a'),$$
(1)

where Pr(S'|s, a) is the probability of transiting to state set S' if action a is taken in state s. If  $Q^*$  is known, then the optimal policy is simply to select at each state the action with the minimum  $Q^*$  value, i.e.  $\pi^*(s) = \arg \min_{a \in A(s)} Q^*(s, a)$ .

The above one-to-many state transitions have already been noticed and handled in Lagoudakis and Littman (2000, 2001), where RL is applied to learn policies for recursive algorithm selection and choosing branching literals in the #DPLL procedure for solving SAT problems (with the requirement of finding all solutions). More specifically, Qlearning is used to learn a linearly parameterized function  $Q_w$  as the estimation of  $Q^*$ . Given a transition (s, a, S') with cost r(s, a), the parameters w are updated using the following target:

$$y = r(s, a) + \gamma \sum_{s' \in S'} \min_{a' \in A(s')} Q_{\mathbf{w}}(s', a').$$
(2)

However, the learning mechanisms in Lagoudakis and Littman (2000, 2001) are not suitable for our situation. The key difficulty is that, we do not know S' until the search algorithm backtracks from s. Consider again the example in Fig. 1(a), we know that  $s^E$  has four children at most, but only when backtracking from  $s^E$  can we know that only two of them are needed to be explored. This is not an issue for #DPLL because the algorithm needs to visit all the child nodes eventually. However, in our case, learning must be delayed until backtracking, when the complete transition (s, a, S') and its cost r(s, a) are available so that the target can be computed. This is not desirable because it slows down the learning process, and requires additional engineering efforts to store information for all the pending states.

To make learning faster and more convenient, we intend to achieve bootstrapping using simple transitions (s, a, s') and avoid the need of storing pending states. To this end, we stick to the one-to-one state transition in the typical MDP. More specifically, for state s and action a, we consider the multiple child states in S' as samples drawn from the transition dynamics Pr(s'|s, a) = 1/|S'| for all  $s' \in S'$ , where Pr(s'|s, a)is the probability that the environment will transit to s' if a is taken in s. In this way, we can use the typical Bellman optimality equation to express an optimal action-value function that are different from that in Eq. (1):

$$\hat{Q}^*(s,a) = r(s,a,s') + \gamma \sum_{s'} Pr(s'|s,a) \min_{a' \in A(s')} \hat{Q}^*(s',a').$$
(3)

An advantage of  $\hat{Q}^*$  is that, we can easily apply Q-learning to learn an estimate  $\hat{Q}$  of it by bootstrapping according to simple transition (s, a, s') and cost r(s, a, s'), since they are immediately known after a child node s' is created. Note that the learned  $\hat{Q}$  function is only used to make decision for a given state s when it is visited for the first time, i.e. when the left most branch is created. We denote such action as  $a(s) = \arg \min_{a \in A(s)} \hat{Q}^*(s, a)$ . For other branches, i.e. those created upon backtracking, the same action is imposed, instead of chosen according to  $\hat{Q}$ , so that the requirement of backtracking search is satisfied. This is natural for the testing phase, since a fixed  $\hat{Q}$  always predicts the same value for given s and a. But for training,  $\hat{Q}$  keeps changing, hence we need to enforce the same action for transitions to all the following states in S'.

Note that the optimal policies  $\hat{\pi}^*(s)$  and  $\pi^*(s)$  derived from  $\hat{Q}^*$  and  $Q^*$  are different. In fact,  $\hat{\pi}^*(s)$  minimizes the expected cost of reaching a leaf node in the subtree rooted from *s*. This is aligned with the well-known "fail-first" principle (Haralick and Elliott, 1980) in designing CSP searching strategy, which suggests to reach the leaf nodes as quick as possible. In Section 5, we will show that this intuition is verified: along with the progress of learning an estimate of  $\hat{Q}^*$ , the search tree size also decreases though it is not the direct objective to be optimized. In the following section, we will design a parameterized function  $\hat{Q}_w$  to estimate  $\hat{Q}^*$  by using a deep neural network.

#### 4.2. GNN based representation

To parameterize  $\hat{Q}$ , we need to find a way to represent *s* and *a*. Recall that *s* is a CSP instances or subinstance which can be described

<sup>&</sup>lt;sup>2</sup> From this point on, we will omit the dependency of S' on s and a for brevity.



Fig. 2. Illustration of the Q network. The outputs of  $x_3$  and  $x_5$  are masked out since they are bounded.

as a constraint network, and *a* is an unbounded variable. For binary CSP, i.e. the arity |scp(c)| = 2 for all constraint  $c \in C$ , the underlying constraint network can be viewed as a graph with the variables being vertices and constraints being edges. Such a graph can be naturally represented by GNN frameworks. Essentially, a GNN learns a vector representation, or embedding, for each vertex in a given graph by iteratively performing embedding aggregation among neighboring vertices (Xu et al., 2019). But in general, the arity of CSP constraints could be larger than 2, meaning that the underlying structure is a *hypergraph*, with the constraints being hyperedges. To effectively represent the constraint network, below we design a GNN variant that learns embeddings for both the vertices and hyperedges.

Given a constraint network P, let  $\mathcal{N}_c(j) = \{i | x_i \in scp(c_j)\}$  be the indexes of variables that are in the scope of a constraint  $c_j$ , and  $\mathcal{N}_v(i) = \{j | x_i \in scp(c_j)\}$  be the indexes of constraints where a variable  $x_i$  is involved in. The current status of variables and constraints are characterized by raw feature vectors  $X_i$  and  $C_j$ , each with dimension  $p_v$  and  $p_c$ . Our GNN computes a p-dimensional embedding  $\mu_i$  and  $v_j$  for each variable  $x_i \in \mathcal{X}$  and constraint  $c_j \in C$ .<sup>3</sup> These embeddings are first initialized by linearly transforming the respective raw feature vectors, i.e.  $\mu_i^{(0)} = X_i \mathbf{w}_v$  and  $v_i^{(0)} = C_i \mathbf{w}_c$ , where  $\mathbf{w}_v \in \mathbb{R}^{p_v \times p}$  and  $\mathbf{w}_c \in \mathbb{R}^{p_c \times p}$  are learnable parameters. Then we update these embeddings by performing K iterations of embedding aggregation operations among the variables and constraints, based on the underlying hypergraph structure. More specifically, in each iteration  $k = 1, \ldots, K$ , we perform the following steps:

• Embedding of each constraint  $c_j$  is first updated by aggregating embeddings from the related variables in  $\mathcal{N}_c(j)$ . More specifically, we use element-wise summation as the aggregation function, the result of which is fed into a Multilayer Perceptron (MLP) MLP<sub>v</sub> to get the updated embedding of  $c_j$ , along with its embedding in the previous round and its raw feature vector. This procedure is shown as follows:

$$\boldsymbol{\nu}_{j}^{(k)} \leftarrow \mathrm{MLP}_{\boldsymbol{\nu}}\left[\sum_{i \in \mathcal{N}_{c}(j)} \boldsymbol{\mu}_{i}^{(k-1)} : \boldsymbol{\nu}_{j}^{(k-1)} : \boldsymbol{C}_{i}\right], \tag{4}$$

where  $[\cdot : \cdot]$  is the concatenation operator. The input and output dimension of MLP<sub>v</sub> are  $2p + p_c$  and p, respectively.

• Embedding of each variable  $x_i$  is updated by aggregating embeddings of the related constraints in  $\mathcal{N}_v(i)$ , based on similar procedure shown below:

$$\mu_i^{(k)} \leftarrow \mathrm{MLP}_c \left[ \sum_{j \in \mathcal{N}_v(i)} \nu_i^{(k)} : \mu_i^{(k-1)} : X_i \right].$$
(5)

The input and output dimension of  $\text{MLP}_c$  are  $2p + p_v$  and p, respectively.

To parameterize  $\hat{Q}_w$ , we represent the current state *s* by performing graph-level pooling using element-wise summation of all the variable

embeddings after iteration *K*, i.e.  $\sum_{i=1}^{n} \mu_i^{(K)}$ , similar to Khalil et al. (2017a). Then we concatenate the embedding representations of the graph and corresponding action *a*, and feed it into another MLP to get  $\hat{Q}_w$  as follows:

$$\hat{Q}_{\mathbf{w}}(s,a) = \mathrm{MLP}_{q} \left[ \sum_{i=1}^{n} \mu_{i}^{(K)} : \mu_{a}^{(K)} \right],$$
(6)

where  $MLP_q$  has input and output dimension of 2p and 1, respectively.

The raw features of variables and constraints in a state *s* are summarized below. For each variable  $x_i$ , we use its current domain size  $|d(x_i, s)|$  and a binary indicator  $b(x_i, s)$  specifying whether it is bounded as the raw features, hence the vector dimension  $p_v = 2$ . For each constraint  $c_j$ , its raw feature vector contains: (1) the number of unbounded variables  $ub(c_j, s)$ , and (2) the current constraint tightness  $1 - |rel(c_j, s)|/D(c_j, s)$ , where  $|rel(c_j, s)|$  is the number of currently allowed tuples and  $D(c_j, s) = \prod_{x_i \in scp(c_j)} |d(x_i, s)|$  is the product of current domain sizes of the involved variables. The dimension of constraint feature vector is  $p_c = 2$ .

The computation process of the Q network is visualized in Fig. 2. In the implementation, to make full use of the parallel computing power of GPU, we collect the raw features of all the variables and constraints into two matrices  $\mathbf{X} \in \mathbb{R}^{n \times p_v}$  and  $\mathbf{C} \in \mathbb{R}^{e \times p_c}$  where each row is the corresponding raw feature  $X_i$  and  $C_j$ , and feed them into the network. In this case, the computations in Eqs. (4)–(6) can be done in parallel for all the variables and constraints, in the form of matrix calculations. To be more specific, let  $\boldsymbol{\mu}^{(k)} \in \mathbb{R}^{n \times p}$  and  $\boldsymbol{\nu}^{(k)} \in \mathbb{R}^{e \times p}$  be the embeddings of all the variables and constraints after iteration k. Then the embedding update in Eqs. (4) and (5) can be written as:

$$\mathbf{v}^{(k)} \leftarrow \mathrm{MLP}_{v}\left[\mathbf{A}\boldsymbol{\mu}^{(k-1)}: \mathbf{v}^{(k-1)}: \mathbf{C}\right],\tag{7}$$

$$\boldsymbol{\mu}^{(k)} \leftarrow \mathrm{MLP}_c \left[ \mathbf{A}^\top \boldsymbol{\nu}^{(k)} : \boldsymbol{\mu}^{(k-1)} : \mathbf{X} \right],$$
(8)

where **A** is a  $e \times n$  binary matrix representing the adjacency relations between variables and constraints, and  $\mathbf{A}_{ji}$  is 1 if  $x_i \in scp(c_j)$  and 0 otherwise. Note that we implement **A** as a sparse matrix, and the number of non-zero values in **A** is  $S = \sum_{j=1}^{e} scp(c_j)$ . The matrix form of Eq. (6) is a bit different from its definition. Specifically, we compute the Q-values for all the variables, and mask out the results of bounded variables (i.e. set the Q-value to  $\infty$ ) such that only legal actions in A(s)will be selected. In this case, Eq. (6) can be written as:

$$\hat{\mathbf{Q}}_{\mathbf{w}}(s) = \mathrm{MLP}_{q} \left[ \mathbf{I} \boldsymbol{\mu}^{(K)} : \boldsymbol{\mu}^{(K)} \right], \tag{9}$$

where **I** is a  $n \times n$  all-one matrix.

Here we briefly analyze the time complexity of the Q-network, for which the major computation is on matrix multiplications. To simplify the analysis, we consider the embedding dimension p and raw feature dimensions  $p_v$  and  $p_c$  as constant. Since **A** is sparse, the complexity of Eq. (4) is  $O(Sp + ep(2p + p_c)) = O(S + e)$ , where O(Sp) is for sparse-dense matrix multiplication and  $O(ep(2p + p_c))$  is for dense matrix multiplications in MLP<sub>v</sub>. Similarly, the computation of Eq. (5) is in O(S+n), and the complexity of K iterations is O(K(S + n + e)). Computation of the Q-values in Eq. (6) is in  $O(n^2p + n(2p)) = O(n^2)$ , and hence the overall time complexity of the Q-network is  $O(K(S + n + e) + n^2)$ .

<sup>&</sup>lt;sup>3</sup> Here we use the same embedding dimension for variables and constraints for simplicity; but in general they could have different dimensions.

The above representation scheme inherits the nice property of GNN, i.e. the same model and set of parameters can process instances with arbitrary size and constraint arity, because all the trainable parameters are shared across all its inputs (e.g.  $\text{MLP}_v$  is *shared* across all constraints). This effectively enables generalizing models trained on small instances to larger ones. In the next section, we describe our algorithm for training the RL agent.

#### 4.3. Training algorithm

Our training algorithm is designed based on Double Deep Q-Network (DDQN) (Hasselt et al., 2016). It maintains two networks, i.e. the online network  $\hat{Q}_w$  and target network  $\hat{Q}_{\bar{w}}$ . More specifically,  $\hat{Q}_{\bar{w}}$  is a periodical copy of  $\hat{Q}_w$ . At each state *s*, the RL agent selects an action  $a_t$  using the  $\epsilon$ -greedy strategy according to the online network, and the experience (s, a, s', r, T) is added to an experience reply buffer  $\mathcal{R}$  with memory size  $\mathcal{M}$ , where T = T(s') is a binary indicator of whether s' is a terminal state. Then a mini-batch of  $\mathcal{B}$  transitions is sampled from  $\mathcal{R}$  to update the parameters of the online network  $\hat{Q}_w$  by performing gradient decent to minimize the squared loss between  $\hat{Q}_w(s, a)$  and the following target:

$$y = r + \gamma \hat{Q}_{\bar{\mathbf{W}}} \left( s', \underset{a' \in A(s')}{\arg\min} \hat{Q}_{\mathbf{W}}(s', a') \right).$$
(10)

Note that the above target computation is only applicable when the state s' is non-terminal. For the terminal ones, the target is simply y = r.

## Algorithm 1 DDQN for learning variable ordering heuristic

Initialize the experience replay to capacity  $\mathcal{M}$ for episode e = 1 to  $\mathcal{N}$  do Draw a CSP instance  $P \sim \mathbb{D}$  $\mathcal{T} \leftarrow 0$ while  $T < T_{max}$  and *P* is not solved yet **do** Observe state s, if s has been visited then Choose a = a(s)else Choose *a* as follows:  $a(s) = \begin{cases} \text{randomly choose from } A(s) \\ \arg\min_{a \in A(s)} \hat{Q}_{\mathbf{w}}(s, a) \end{cases}$ w.p. *e* otherwise end if Execute *a*, observe s' and T(s')Store (s, a, s', 1, T(s')) in  $\mathcal{R}$ Randomly sample a minibatch from R For each sampled experience, compute the target: If s' is terminal r y = $\int$  use Eq. (10) otherwise Perform a gradient descent step to update w  $\mathcal{T} \leftarrow \mathcal{T} + 1$ if P is solved then Break while end if end while For every  $e_{\mu}$  episodes, set  $\bar{\mathbf{w}} = \mathbf{w}$ end for

Our training algorithm is shown in Algorithm 1. The agent is trained for  $\mathcal{N}$  episodes, during each of them the agent tries to solve a CSP instance drawn from the distribution  $\mathbb{D}$ . Due to the intractability of CSP, it is possible that solving an instance requires a very large number of steps, i.e. state transitions, especially in the beginning stage of learning when  $\epsilon$  is large and the quality of policy is low. Though we can let the agent finish solving an instance, this is not desirable because the agent's experience may be limited to a small number of instances for a long time. Therefore, a special design here is that we set a cutoff limit of  $\mathcal{T}_{max}$  steps (equivalent to the maximum number of search nodes) to limit the effort spent by the agent on one instance. Note that the terminal indicator T(s') of an experience is true only when s' corresponds to a leaf node. For those s' terminated due to reaching  $\mathcal{T}_{max}$ , T(s') is still false. This is to ensure that the target is correctly computed: the actual cost of a state s' terminated by  $\mathcal{T}_{max}$  is not 0, since more nodes under it need to be visited to solve the subinstance in s'. This corresponds to the partial-episode bootstrapping method in Pardo et al. (2018). In the experiments, we will show that even a large number of instances hit the cutoff limit in the early stage, the agent can still learn high-quality policy that solves all testing instances.

#### 5. Computational experiments

In this section, we conduct a series of experiments to test the proposed approach. We first introduce the setup of our experiments, then present the training and testing results on small-sized instances, and finally report the generalization performance on larger instances, as well as some analysis of the execution time.

#### 5.1. Experimental setup

**Instance generation.** The CSP instance used in our experiments are generated using the well-known and widely used random CSP generator, model RB (Xu et al., 2007). It takes 5 parameters  $\langle m, n, \alpha, \beta, \rho \rangle$  as input to generate a CSP instance, the meanings of which are listed below:

- $m \ge 2$  is the arity of each constraint;
- $n \ge 2$  is the number of variables;
- $\alpha > 0$  specifies *d*, which is the domain size of each variable, and  $d = n^{\alpha}$ ;
- $\beta > 0$  specifies *e*, which is the number of constraints, and  $e = \beta \cdot n \cdot \ln n$ ;
- *ρ* ∈ (0, 1) specifies the constraint tightness, and *ρ* · *d<sup>k</sup>* is the number of disallowed tuples for each constraint.

Each unique combination of the above parameters specifies a class of CSP instances, which can be considered as the distribution  $\mathbb{D}$ . The CSP classes used in our experiments are all situated at the *phase transition* thresholds, which are combinations of parameters that result in the *hardest* instances. A nice theoretical property of model RB that makes it more preferable than other random CSP models is that, it can guarantee exact phase transitions and instance hardness at the threshold (Xu et al., 2007). We test our approach for two types of distributions with binary and 3-ary constraints, denoted as  $\mathbb{D}_1(n) = \langle 2, n, 0.7, 3, 0.21 \rangle$  and  $\mathbb{D}_2(n) = \langle 3, n, 0.7, 2.5, 0.24 \rangle$ , respectively. With different *n*, we have CSP classes with different sizes. In our experiments, we choose *n* from {15, 25, 30, 40} and {10, 15, 20, 25} for  $\mathbb{D}_1(n)$  and  $\mathbb{D}_2(n)$ , respectively, since higher constraint arity generally leads to harder instances.

Implementation details. For our GNN model, we instantiate it by setting the embedding dimension p = 128, and all MLPs have L = 3 layers with hidden dimension 128 and rectified linear units as activation function. The embeddings are updated for K = 5 iterations. We train the RL agent for  $\mathcal{N} = 1000$  episode, i.e. solving 1000 instances drawn from distribution  $\mathbb D,$  with the cutoff step limit  $\mathcal T_{max}$  = 10000. During training, another 200 instances drawn from  ${\ensuremath{\mathbb D}}$  are used to validate performance of the agent's policy. The discount factor  $\gamma$  is set to 0.99. For exploration, the value of  $\epsilon$  is set to 1 in the beginning, and linearly annealed to 0.05 in the first 20000 steps. We use the Adam optimizer to train the neural network, with a constant learning rate  $\eta = 0.00005$ and mini-batch size B = 128. The size of experience replay is  $\mathcal{M} = 0.1$ M. The frequency of updating the target network is  $e_{\mu} = 100$ . For testing, we set a cutoff limit of  $5 \times 10^5$  search nodes for our policies and all baseline heuristics. The neural architecture and hyperparameters are empirically tuned on small sized instances. More details will be given in Section 5.4. Our approach is implemented in C++ on top of the source code of Google OR-Tools,<sup>4</sup> a state-of-the-art CSP solver, which employs the binary branching strategy. The GNN architecture and training algorithm is implemented based on the source code<sup>5</sup> of Khalil et al. (2017a). The hardware we used is a workstation with Intel Core i9-9920x CPU and one NVIDIA RTX 2080Ti GPU (11 GB memory). Our code is publicly available at https://github.com/songwenas12/csp-drl.

**Baselines.** We compare the trained policies with four classic handcrafted variable ordering heuristics that are representative and commonly used in many CSP solvers. The first two are embedded in the underlying solver OR-Tools (other embedded heuristics are rather uncompetitive in the experiments hence ignored here):

- MinDom (Haralick and Elliott, 1980), which selects the variable with the minimum current domain size (Dom). Despite its simplicity, this heuristic is effective and popular in practice, and is implemented in almost all solvers.
- Impact-based heuristic (Impact) (Refalo, 2004), which selects the variable that can lead to an assignment with the maximum reduction of search space, i.e. impact. Impact is considered as one of the state-of-the-art variable ordering heuristics (Li et al., 2016), and is implemented as the default search strategy in OR-Tools. Here we use its default configuration where the impact of a variable is measured by summing the impact of each value in its current domain, and the decision logic is to select the variable with the maximum impact and the value with the minimum impact.

Note that even with its embedded heuristics, OR-tools is already a very competitive solver and has won the MiniZinc Challenge (Stuckey et al., 2014) several times. Nevertheless, we implement the below two heuristics for comparison:

- Dom/Ddeg (Bessiere and Régin, 1996), which improves MinDom by taking the dynamic degree (Ddeg) of a variable into account, and selects the variable with the minimum ratio between Dom and Ddeg. To compute the Ddeg of a variable  $x_i$ , this heuristic first identifies the set of constraints involving  $x_i$ , i.e.  $C(x_i) = \{c_j \in C | x_i \in scp(c_j)\}$ ; then it removes from  $C(x_i)$  those constraints that involve no unbounded variables, and uses the cardinality of  $C(x_i)$ as the Ddeg value of  $x_i$ .
- Dom/Tdeg (Li et al., 2016), which is a recently proposed heuristic that incorporates constraint tightness that is also used in our method as a feature. It replaces the Ddeg in Dom/Ddeg by the dynamic tightness degree (Tdeg), defined as the summation of tightness of constraints in  $C(x_i)$ , instead of the cardinality as in Ddeg.

Except Impact which has its own value ordering heuristic, we apply lexicographical ordering for our approach (denoted as DRL) and the other baselines MinDom, Dom/Ddeg and Dom/Tdeg to select the next value.

## 5.2. Training and testing on small and medium sized instances

In this section, we discuss the performance of the RL agent during training. More specifically, we train the agent on small and medium sized binary and 3-ary instance distributions including  $\mathbb{D}_1(15)$ ,  $\mathbb{D}_1(25)$ ,  $\mathbb{D}_2(10)$  and  $\mathbb{D}_2(20)$ , respectively. We use two measures to evaluate the agent's policies, including (1) the number of search nodes, which is the objective we try to minimize, and (2) the number of failures (i.e. deadends), which is a measure of the ability to "fail first" (Beck et al., 2004) and hence reflects the performance with respect to the objective we defined in Eq. (3).

In Figs. 3(a) and 3(b), we plot the agent's training performance on the two smallest distributions, i.e.  $\mathbb{D}_1(15)$  and  $\mathbb{D}_2(10)$ , with respect to the average values of the above two measures on the 200 validation instances. For these two distributions, all instances are successfully solved within the 10000 cutoff step limit. As can be observed, the agent's performance significantly improved during training. For example, on  $\mathbb{D}_1(15)$ , the agent needs to visit over 150 search nodes with more than 70 times of failure on average in the beginning stage to solve an instance. With the increase of training episodes (i.e. the number of training instances), the two measures significantly drops to 22-23 search nodes and 8-9 failures. Similar trends also exist in Fig. 3(b). However, the training process on  $\mathbb{D}_1(10)$  is more fluctuated, and requires more episodes to converge to a policy with 50-100 search nodes and 30-50 failures. This indicates that though having smaller number of variables, solving instances drawn from  $\mathbb{D}_2(10)$  is harder than solving those from  $\mathbb{D}_2(15)$ , in terms of both learning and solving.

Figs. 3(c) and 3(d) show the training curves on medium sized distributions  $\mathbb{D}_1(25)$  and  $\mathbb{D}_2(15)$ . Clearly, instances drawn from these two distributions are much harder to solve. Despite the large number of search nodes and failures, the agent even cannot solve many validation instances within the cutoff limit of 10 000 nodes in the early stage. For example, on the hardest distribution  $\mathbb{D}_2(15)$ , there are still 67.5% (135 out of 200) validation instances cannot be solved after 100 episodes of training. Nevertheless, the agent can still drastically improve its performance. For example, on  $\mathbb{D}_2(15)$ , the number of search nodes significantly drops from nearly 9000 to about 830, and all validation instances are successfully solved. Note that on the same scale, the cures in Figs. 3(c) and 3(d) are more fluctuated than their counterparts in Figs. 3(a) and 3(b).

Fig. 3 demonstrates the effectiveness of our design in several aspects. Firstly, for the learning tasks on distributions with different features and hardness, the agent is able to learn a variable ordering policy from scratch by itself, without the need of supervision. Secondly, though the agent is optimizing an alternative objective defined in Eq. (3), with the progress of learning, the search tree size also decreases with almost identical trend. Finally, the large number of cutoff instances in the early stages does not affect the training process to converge to high-quality policies.

We then evaluate the quality of the trained policies by comparing them with the baseline heuristics on four test sets, each with 500 instances drawn from the corresponding distributions used in training. We use the best policy (with respect to the number of search nodes) to obtain the results of our approach DRL. In these experiments, all the testing instances are successfully solved by all heuristics within the cutoff limit of  $5 \times 10^5$  search nodes. The results are summarized in Table 1, where the column "Average" is the mean number of search nodes (failures) for each heuristic on the 500 testing instances, and the column "Reduction" is the percentage reduction in the average number of search nodes (failures) that our policies achieved compared with each baseline. We can observe that for all the four distributions, the policies trained by our approach outperform the two baseline heuristics embedded in OR-Tools, i.e. MinDom and Impact, by a large margin. The reason why Impact performs not quite well is probably because random CSP instances do not have very strong structure (Correia and Barahona, 2008). For the other two baselines, our method consistently outperforms Dom/Ddeg on all the four distributions. In terms of Dom/Tdeg, our method performs on par with it on the two small distributions  $\mathbb{D}_1(15)$  and  $\mathbb{D}_2(10)$ , since the difference between DRL and Dom/Tdeg is not statistically significant.<sup>6</sup> However, on the two larger and harder distributions  $\mathbb{D}_1(25)$  and  $\mathbb{D}_2(15)$ , DRL significantly outperforms Dom/Tdeg with around 10% reduction in the number of nodes and failures, and is statistically significant (for # nodes and failures, pvalue is  $2.94 \times 10^{-10}$  and  $3.52 \times 10^{-10}$  on  $\mathbb{D}_1(25)$ , and  $5.22 \times 10^{-12}$  and

 <sup>&</sup>lt;sup>4</sup> https://github.com/google/or-tools Note that our implementation is based on the original CP solver, instead of the CP-SAT solver.
 <sup>5</sup> https://github.com/Hanjun-Dai/graph comb opt.

<sup>&</sup>lt;sup>6</sup> We use the Wilcoxon signed-rank test with p < 0.001 for all the statistical significant tests in this paper.

![](_page_8_Figure_2.jpeg)

Fig. 3. Training Performance (Evaluated every 50 training instances).

 Table 1

 Test results on the same distribution used in training.

$\mathbb{D}$	Heuristic	# Search nodes		# Failures	
		Average	Reduction	Average	Reduction
	DRL	22.21	-	8.60	-
	Dom/Tdeg	22.81	-	8.91	-
$\mathbb{D}_1(15)$	Dom/Ddeg	23.05	3.64%	9.02	4.66%
	MinDom	33.57	33.85%	14.15	39.22%
	Impact	272.51	91.85%	134.99	93.63%
	DRL	56.65	-	26.60	-
	Dom/Tdeg	57.82	-	27.12	-
$\mathbb{D}_2(10)$	Dom/Ddeg	59.98	5.55%	28.27	5.91%
	MinDom	100.46	43.61%	48.40	45.04%
	Impact	514.09	88.98%	267.75	90.07%
	DRL	291.30	-	141.92	-
	Dom/Tdeg	320.19	9.02%	156.26	9.18%
$\mathbb{D}_1(25)$	Dom/Ddeg	347.78	16.24%	170.06	16.54%
	MinDom	799.54	63.57%	395.82	64.14%
	Impact	69885.05	99.58%	37526.49	99.62%
	DRL	972.45	-	483.58	-
	Dom/Tdeg	1084.81	10.36%	539.80	10.41%
$\mathbb{D}_2(15)$	Dom/Ddeg	1143.85	14.98%	569.25	15.05%
	MinDom	2537.24	61.67%	1265.90	61.80%
	Impact	48941.93	98.01%	26267.43	98.16%

 $4.87 \times 10^{-12}$  on  $\mathbb{D}_2(15)$ ). This key observation indicates that our method could be more effective on harder problems. A possible explanation is that, the search tree for harder problems are larger, hence the space for possible improvement is also larger.

#### 5.3. Generalizing to larger instances

As mentioned previously, our GNN based representation enables generalizing the trained models to larger instances that have never been seen by the agent during training. In this section, we conduct experiments to evaluate the generalization performance. Specifically, for distributions  $\mathbb{D}_1(n)$ , we run the policy trained on  $\mathbb{D}_1(25)$  on larger distributions with  $n \in \{30, 40\}$ ; for the distributions of 3-ary CSP  $\mathbb{D}_2(n)$ , the policy trained on  $\mathbb{D}_2(15)$  is evaluated on distributions with  $n \in \{20, 25\}$ . Similar as in the previous section, we randomly sample 500 instances from each distribution as the test set. In Tables 2 and 3, we summarize the results on binary and 3-ary CSP distributions  $\mathbb{D}_1(n)$  and  $\mathbb{D}_2(n)$ , respectively. In these tables, values in the columns "Average" are computed using all results on the 500 testing instances,

but those in columns "Reduction" are computed based on the instances that are solved by both DRL and the comparing baseline. The columns "# Cutoff" show the number of instances that are not solved by each heuristic within the cutoff limit of  $5 \times 10^5$  nodes. Note that for those experiments that a majority ( $\geq 250$ ) of instances are cut off, we do not report the reduction values.

In Table 2, we can observe that with the increase of *n*, the hardness of solving the instances grows rapidly for all heuristics, which shows the exponential complexity of CSP. For all experiments in this table, our policy trained on  $\mathbb{D}_1(25)$  still significantly outperforms all the baselines, with even larger reduction compared with the corresponding results in Table 1. Moreover, the reduction tends to be more significant on instances with larger n which are harder to solve. Our conjecture for this behavior is that, the performance of the trained policy degrades as the increase of problem size which is common for existing deep learning based approaches, e.g. Khalil et al. (2017a) and Kool et al. (2019). However, for larger instances, the spaces that the learned policy can improve over classic heuristics also become larger, which surpass the effect of performance degradation. For Table 3 which summarizes the results for 3-ary distributions, we can make almost the same observations as those for Table 2. Moreover, we notice that the reductions over the baseline heuristics in these experiments on 3-ary distributions are more prominent than those on binary distributions, especially for the most competitive baseline Dom/Tdeg.

To have a more fine-grained analysis and visualization of the aggregate results in Tables 2 and 3, we show the cactus plots of our method and baselines on the hardest binary and 3-ary distribution  $\mathbb{D}_1(40)$  and  $\mathbb{D}_2(25)$  in Fig. 4 (Impact is ignored due to the large number of cutoff instances). A cactus plot shows the increase of solved instances with the increase of solving resource bound (cutoff node limit here), and the lower and more to the right a curve is, the better the corresponding method. We can see that with the same cutoff node limit, our method almost consistently solves more instances than all baselines. Such improvement is more significant on the harder distribution  $\mathbb{D}_2(25)$ . To further validate the advantage of our method on the per-instance basis, we plot the pairwise comparisons against Dom/Tdeg and Dom/Ddeg on  $\mathbb{D}_1(40)$  and  $\mathbb{D}_2(25)$  in Fig. 5, where each dot is an instance and the coordinates are the corresponding search tree size of our method and the baseline. We can see that on both distributions, most dots are distributed above the diagonal (the dashed line), meaning that our method solves the corresponding instances with fewer nodes than the baselines.

To summarize, the above analysis shows that the policies trained by our approach have good generalization ability, and tend to perform

#### Table 2

Generalization	results	on	binary	CSP	distributions.
----------------	---------	----	--------	-----	----------------

D	Heuristic	# Search Node	s	# Failures	# Cutoff	
		Average	Reduction	Average	Reduction	
	$DRL(\mathbb{D}_1(25))$	1237.93	-	614.44	-	-
	Dom/Tdeg	1350.92	8.36%	670.93	8.42%	-
$\mathbb{D}_1(30)$	Dom/Ddeg	1523.92	18.77%	757.38	18.87%	-
	MinDom	4160.78	70.25%	2075.69	70.40%	-
	Impact	318862.66	-	171029.08	-	255
	$DRL(\mathbb{D}_1(25))$	23684.04	-	11836.26	-	-
	Dom/Tdeg	26861.89	11.83%	13425.07	11.83%	-
$\mathbb{D}_1(40)$	Dom/Ddeg	31807.81	25.54%	15897.96	25.55%	-
	MinDom	136405.52	81.38%	68196.94	81.39%	13
	Impact	491872.80	-	263666.30	-	492

Note: Reduction values are computed using instances solved by both DRL and the corresponding baseline method.

#### Table 3

Generalization	results	on	3-arv	CSP	distributions.
Jeneranzation	resuits	on	J-ary	COL	uisti ibutions.

$\mathbb{D}$	Heuristic	# Search node	# Search nodes		# Failures		
		Average	Reduction	Average	Reduction		
	$DRL(\mathbb{D}_2(15))$	17421.14	-	8707.31	-	-	
	Dom/Tdeg	20598.00	15.42%	10295.76	15.43%	-	
$\mathbb{D}_2(20)$	Dom/Ddeg	22113.32	21.22%	11053.37	21.22%	-	
	MinDom	73781.04	75.46%	36887.05	75.46%	3	
	Impact	462299.49	-	248075.41	-	436	
	$DRL(\mathbb{D}_2(15))$	287406.14	-	143701.41	-	153	
	Dom/Tdeg	322161.17	18.74%	161079.64	18.74%	191	
$D_{2}(25)$	Dom/Ddeg	330373.63	22.08%	165186.04	22.08%	203	
-	MinDom	438152.80	-	219079.34	-	391	
	Impact	499595.53	-	268061.19	-	498	

Note: Reduction values are computed using instances solved by both DRL and the corresponding baseline method.

![](_page_9_Figure_10.jpeg)

Fig. 4. Cactus plots of our method and baselines Dom/Tdeg, Dom/Ddeg, and MinDom.

better on harder instances with larger sizes and higher constraint arities, exhibiting even larger improvements than testing on instances of the same size as used in training. To support our observations

Table 4 Statistical significant test results (p value) for DRL and Dom/Tdeg.

# Search nodes	# Failures
2.24×10 <sup>-12</sup> 3.71×10 <sup>-20</sup>	2.28×10 <sup>-12</sup> 3.66×10 <sup>-20</sup>
<b>3.64</b> ×10 <sup>-39</sup> <b>3.43</b> ×10 <sup>-13</sup>	3.65×10 <sup>-39</sup> 3.43×10 <sup>-13</sup>
	# Search nodes 2.24×10 <sup>-12</sup> 3.71×10 <sup>-20</sup> 3.64×10 <sup>-39</sup> 3.43×10 <sup>-13</sup>

and conclusion, we conduct statistical significant test to compare the performance of DRL and Dom/Tdeg.<sup>7</sup> The p values are listed in Table 4, showing that all results are statistically significant (p < 0.001).

# 5.4. Impact of different neural network architecture

As mentioned previously, the network architecture and hyperparameters are tuned by performing grid search on small-sized instances, following the common practice in the literature. Here we show the impact of two hyperparameters, i.e. the number of GNN iteration K and the number of MLP layers L, which we find to be crucial for the performance. We set  $K \in \{1, 3, 5, 7\}$  and  $L \in \{2, 3, 4, 5\}$ , and perform training on the simplest distribution  $\mathbb{D}_1(15)$  for each combination of *K* and *L*. We also test the trained network on larger distribution  $\mathbb{D}_1(20)$  to examine the generalization capability. As can be observed in Table 5, all combinations perform similarly on the training distribution. However, the generalization performance is significantly different. Specifically, with the increase of *K*, there is a notable trend that the search tree size on  $\mathbb{D}_1(20)$  decreases, showing its great impact on generalization. This could be because with more iterations of aggregation and update, the variable and constraint embeddings could include richer information from neighbors located farther away on the constraint network. For each K, the increase of L seems to be harmful, probably because of

<sup>&</sup>lt;sup>7</sup> Comparison to other baselines are also statistically significant, but the results are omitted here for brevity.

![](_page_10_Figure_1.jpeg)

Fig. 5. Pairwise comparison of Number of Search Nodes against Dom/Tdeg and Dom/Ddeg.

overfitting. The best performing combination is K = 5 and L = 3, which is also the configuration we used in the experiments.

#### 5.5. Execution time analysis

We finally analyze the execution times of all the heuristics, which are listed in Table 6 (some results of Impact are ignored because a majority of instances are cut off). We can observe that the total execution times of DRL (shown in the "Total" column) are much longer than those of the baselines. As shown in the "Infer" column, almost over 90% of the execution time is spent in GNN inference. In contrast, the times for variable ordering decisions of the baseline heuristics are negligible since the related computation is very simple and fast. With the increase of problem size, the percentage of inference time decreases, showing that more portion of time is spent in "actually" solving the instances (mostly constraint propagation). To measure these efforts, we calculate the execution times of DRL without inference in the "No Infer" column. We can observe that the "No Infer" time of DRL is generally much short than the run time of MinDom and Impact, the two embedded heuristics in OR-tools, and comparable to those of Dom/Ddeg and Dom/Tdeg.

Table 5

Impact	of the	number of	GNN	iteration	K	and	MLP	layers	<i>L</i> .			
K	L	Training	on $\mathbb{D}$	<sub>1</sub> (15)			_	Gener	alization	on	$\mathbb{D}_1(20)$	))

K	L	Training on $\mathbb{D}_1(15)$	)	Generalization on	$\mathbb{D}_{1}(20)$
		# Search nodes	# Failures	# Search nodes	# Failures
	2	25.63	10.35	329.7	162.13
1	3	25.63	10.35	718.81	356.57
1	4	26.51	10.80	190.45	92.15
	5	25.59	10.28	212.71	103.21
	2	25.14	10.12	65.26	29.48
2	3	26.12	10.55	69.94	31.93
5	4	26.16	10.55	117.81	55.69
	5	26.42	10.75	141.10	67.36
	2	25.38	10.15	63.72	28.68
E	3	25.34	10.12	60.96	27.34
5	4	25.08	10.03	61.32	27.49
	5	26.13	10.65	68.07	30.98
	2	25.60	10.25	63.66	28.63
7	3	25.57	10.30	74.86	34.32
/	4	25.09	10.03	67.76	30.65
	5	27.11	10.98	81.71	37.61

Note that it is common in tree search algorithms that high-quality heuristics leading to smaller search tree consume much longer computational time, such as the well-known and effective heuristic Strong Branching for MILP (Applegate et al., 1995). Nevertheless, minimizing search tree size is a valid and well-recognized objective for both traditional and learning-based methods (Liberatore, 2000; Hooker, 1995; Zarpellon et al., 2021). Our objective in this paper is not to compete with traditional heuristics in execution time, but to show that highquality search policies for CSP can be automatically learned in a data-driven way, by a carefully designed deep RL method. We believe the efficiency of our method could be significantly improved with more advanced techniques such as parameter pruning and model compression, but is out of the scope of this paper and is a direction for future research.

Here we show a simple strategy that maintains the advantage of our method in producing smaller search trees, but is comparable and even faster than traditional heuristics in execution time. Specifically, based on the intuition that earlier decisions in the search process are more critical, we perform DNN inference on nodes in the topmost  $\mathcal{K}$  layers of the search tree, and use traditional heuristics for the other nodes. We perform experiments on the hardest distributions  $\mathbb{D}_1(40)$  and  $\mathbb{D}_2(25)$ for binary and 3-ary problems. We choose  $\mathcal{K} \in \{3, 5, 10, 15\}$ , and use the best-performing traditional heuristic Dom/Tdeg for the remaining nodes. Results are summarized in Tables 7 and 8, where the "Ratio" column is the ratio of execution time of DRL to that of Dom/Tdeg, computed on the commonly solved instances. We can observe that this strategy significantly reduces the execution time. With the decrease of  $\mathcal{K}$ , the speed-up is more prominent while the reduction in search tree size becomes smaller, which is an expected trade-off effect. On  $D_1(40)$ , DRL starts to outperform Dom/Tdeg in execution time when  $\mathcal{K} = 5$ , and is 4.3% faster than Dom/Tdeg when  $\mathcal{K} = 3$  while still maintaining 5.43% reduction in the search tree size. On the harder distribution  $D_2(25)$ , DRL starts to be faster than Dom/Tdeg when  $\mathcal{K} = 10$ . The fastest version is DRL ( $\mathcal{K} = 5$ ), which is 7.9% faster than Dom/Tdeg on the solved instances with 3.82% reduction in the search tree size.

#### 6. Discussions and limitations

Experimental results in Section 5 show that in most cases, the variable ordering heuristics learned by our DRL method are able to outperform traditional hand-crafted ones in minimizing the search tree size. This advantage could come from the following important facts. First, in our method, the ordering decisions are made based on a learned Q function which is explicitly optimized towards minimization of the search tree, guided by the surrogate objective defined in Eq. (3).

Table 6

Average execution time for all heuristics (all units are milliseconds unless stated).

D	DRL			Dom/Tdeg	Dom/Ddeg	MinDom	Impact
	Total	Infer	No Infer				
$\mathbb{D}_1(15)$	23.47	98.32%	0.4	0.32	0.26	0.40	4.67
$\mathbb{D}_1(25)$	308.71	91.58%	26.01	23.04	23.70	48.62	2779.89
$\mathbb{D}_1(30)$	1465.66	91.01%	131.77	121.77	120.85	298.5	14339.15
$\mathbb{D}_1(40)$	36820.6	90.02%	3674.38	3881.52	4132.99	16451.9	-
$\mathbb{D}_2(10)$	46.84	97.14%	1.34	1.12	1.07	1.61	5.81
$\mathbb{D}_2(15)$	788.92	93.54%	50.96	47.35	46.01	95.36	1349.70
$\mathbb{D}_2(20)$	16883.9	91.97%	1356.42	1409.26	1371.18	4378.82	-
$\mathbb{D}_2(25)$	318995.48	89.17%	34540.49	35581.39	33515.28	43539.38	-

#### Table 7

Performance of inferring on the topmost  $\mathcal{K}$  nodes on  $\mathbb{D}_1(40)$ .

Heuristic	# Search Nodes		# Cutoff	Time (millis	econds)
	Average	Reduction		Average	Ratio (×)
Dom/Tdeg	26861.89	-	-	3881.52	-
DRL	23684.04	11.83%	-	36820.63	9.486
DRL( <i>K</i> =15)	23327.71	13.16%	-	14245.39	3.670
DRL( <i>K</i> =10)	23967.82	10.77%	-	5840.75	1.505
$DRL(\mathcal{K}=5)$	24983.85	6.99%	-	3793.13	0.977
DRL( <i>K</i> =3)	25402.32	5.43%	-	3698.68	0.953

Note: Reduction and Ratio values are computed using instances solved by both DRL and Dom/Tdeg.

#### Table 8

Performance of inferring on the topmost  $\mathcal{K}$  nodes on  $\mathbb{D}_2(25)$ .

Heuristic	# Search Nodes		# Cutoff	Time (millise	Time (milliseconds)	
	Average	Reduction		Average	Ratio (×)	
Dom/Tdeg	322161.17	-	191	35581.39	-	
DRL	287406.14	18.74%	153	329637.97	8.174	
DRL( <i>K</i> =15)	294922.66	14.81%	159	55643.28	1.651	
DRL( <i>K</i> =10)	302834.59	6.00%	169	35317.73	0.962	
DRL( <i>K</i> =5)	309851.98	3.82%	179	34302.71	0.921	
DRL( <i>K</i> =3)	312273.33	3.07%	181	34463.92	0.933	

Note: Reduction and Ratio values are computed using instances solved by both DRL and  $\mathsf{Dom}/\mathsf{Tdeg}$ .

In contrast, the hand-crafted heuristics are designed based on human experience, which have no direct relation with the search effort. Second, traditional heuristics make decisions based on only simple features (e.g. domain and dynamic degree of each variable). In contrast, for our method, the features extracted by our GNN based scheme is more informative and comprehensive, which provides more accurate solving status so that better decisions could be made. Finally, the size-agnostic property of GNN makes it possible to reuse the knowledge learned on small instances to larger ones. Also, the message passing mechanism of GNN captures the local patterns of the underlying state graph, which could also apply to larger instances.

While showing promising results, our method also has several limitations that need to be improved in the future for practical usage. First, currently we only consider table constraints. While they can theoretically encode any constraints, in practice the constraints could be defined more compactly in intension. One solution to this issue is to extend our method with type-aware constraint representation which could allow heterogeneous edge features, so as to model more constraint types. Second, as we have analyzed in Section 5.5, the inference time of GNN is a bottleneck of our method. Similar situations have also been noticed in other recent works, e.g. Cappart et al. (2021) and Chalumeau et al. (2021). While we have shown a simple strategy in Section 5.5 (inferring GNN on the topmost search nodes) could partially resolve this issue, more principled and rigorous design is needed to achieve desired trade-off between model accuracy and efficiency, such as the hybrid scheme in Gupta et al. (2020).

#### 7. Conclusions and future work

In this paper, we make a preliminary attempt on tackling CSP using deep reinforcement learning. Specifically, we study the problem of how to use DRL to automatically discover a variable ordering heuristic for a given class of CSP instances. We propose a reinforcement learning formulation for this task, which allows the decision making agent to learn from its own solving experiences without the need of supervision. Extensive experiments show that our RL agent can discover variable ordering policies that are better than traditional hand-craft heuristics, in terms of minimizing the search tree size. The learned policies is size agnostic, and can generalize to instances that are larger than those used for training. Moreover, the improvement over traditional heuristics tends to be more significant on larger and harder problems. The framework we designed could also be applied to learn other control policies in backtracking search algorithms, such as value ordering, propagator selection, and determining propagation level. In the future, we plan to improve and extend our method to enhance practical CSP solving in real industrial applications, such as resource scheduling, industrial robot control, and logistics planning.

#### Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

#### Acknowledgements

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from Singapore Telecommunications Limited (Singtel), through Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU). This study is also supported by the National Natural Science Foundation of China under Grant 62102228 and 61803104, and in part by Shandong Provincial Natural Science Foundation under Grant ZR2021QF063, and in part by the A\*STAR Cyber-Physical Production System (CPPS) – Towards Contextual and Intelligent Response Research Program, under the RIE2020 IAF-PP Grant A19C1a0018.

#### References

- Altan, A., Karasu, S., Zio, E., 2021. A new hybrid model for wind speed forecasting combining long short-term memory neural network, decomposition methods and grey wolf optimizer. Appl. Soft Comput. 100, 106996.
- Amizadeh, S., Matusevych, S., Weimer, M., 2019. Learning to solve circuit-SAT: An unsupervised differentiable approach. In: International Conference on Learning Representations.
- Applegate, D., Bixby, R., Chvatal, V., Cook, B., 1995. Finding cuts in the TSP (a preliminary report). Center for Discrete Mathematics & Theoretical Computer Science.
- Balafrej, A., Bessiere, C., Paparrizou, A., 2015. Multi-armed bandits for adaptive constraint propagation. In: International Joint Conference on Artificial Intelligence. pp. 290–296.

- Beck, J.C., Prosser, P., Wallace, R.J., 2004. Trying again to fail-first. In: International Workshop on Constraint Solving and Constraint Logic Programming. Springer, pp. 41–55.
- Behrens, J.K., Lange, R., Mansouri, M., 2019. A constraint programming approach to simultaneous task allocation and motion scheduling for industrial dual-arm manipulation tasks. In: 2019 International Conference on Robotics and Automation. ICRA, IEEE, pp. 8705–8711.
- Bengio, Y., Lodi, A., Prouvost, A., 2020. Machine learning for combinatorial optimization: a methodological tour d'horizon. European J. Oper. Res. 290 (2), 405–421.
- Bessiere, C., Régin, J.-C., 1996. Mac and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 61–75.
- Cappart, Q., Moisan, T., Rousseau, L.-M., Prémont-Schwarz, I., Cire, A.A., 2021. Combining reinforcement learning and constraint programming for combinatorial optimization. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35, no. 5. pp. 3677–3687.
- Chalumeau, F., Coulon, I., Cappart, Q., Rousseau, L.-M., 2021. Seapearl: A constraint programming solver guided by reinforcement learning. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Springer, pp. 392–409.
- Correia, M., Barahona, P., 2008. On the efficiency of impact based heuristics. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 608–612.
- Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.-C., Schaus, P., 2016. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 207–223.
- Ding, J.-Y., Zhang, C., Shen, L., Li, S., Wang, B., Xu, Y., Song, L., 2020. Accelerating Primal Solution Findings for Mixed Integer Programs Based on Solution Prediction. In: Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence. pp. 1452–1459.
- Epstein, S., Petrovic, S., 2017. Learning to solve constraint problems. In: ICAPS-07 Workshop on Planning and Learning.
- Galassi, A., Lombardi, M., Mello, P., Milano, M., 2018. Model agnostic solution of CSPs via deep learning: A preliminary study. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Springer, pp. 254–262.
- Gasse, M., Chételat, D., Ferroni, N., Charlin, L., Lodi, A., 2019. Exact combinatorial optimization with graph convolutional neural networks. In: Advances in Neural Information Processing Systems. pp. 15580–15592.
- Gent, I.P., MacIntyre, E., Presser, P., Smith, B.M., Walsh, T., 1996. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 179–193.
- Gupta, P., Gasse, M., Khalil, E., Mudigonda, P., Lodi, A., Bengio, Y., 2020. Hybrid models for learning to branch. Adv. Neural Inf. Process. Syst. 33, 18087–18097.
- Haralick, R.M., Elliott, G.L., 1980. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence 14 (3), 263–313.
- Hasselt, H.v., Guez, A., Silver, D., 2016. Deep reinforcement learning with double Q-Learning. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 2094–2100.
- He, H., Daume III, H., Eisner, J.M., 2014. Learning to search in branch and bound algorithms. In: Advances in Neural Information Processing Systems. pp. 3293–3301.
- Hooker, J.N., 1995. Testing heuristics: We have it all wrong. J. Heuristics 1 (1), 33–42.
   Kasprzak, W., Szynkiewicz, W., Zlatanov, D., Zielińska, T., 2014. A hierarchical CSP search for path planning of cooperating self-reconfigurable mobile fixtures. Eng.
- Appl. Artif. Intell. 34, 85–98.Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L., 2017a. Learning combinatorial optimization algorithms over graphs. In: Advances in Neural Information Processing Systems. pp. 6348–6358.
- Khalil, E.B., Dilkina, B., Nemhauser, G.L., Ahmed, S., Shao, Y., 2017b. Learning to run heuristics in tree search. In: International Joint Conference on Artificial Intelligence. pp. 659–666.
- Khalil, E.B., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B., 2016. Learning to branch in mixed integer programming. In: Thirtieth AAAI Conference on Artificial Intelligence. pp. 724–731.
- Kool, W., van Hoof, H., Welling, M., 2019. Attention, Learn to Solve Routing Problems! In: International Conference on Learning Representations.
- Lagoudakis, M.G., Littman, M.L., 2000. Algorithm Selection using Reinforcement Learning. In: International Conference on Machine Learning. pp. 511–518.

- Lagoudakis, M.G., Littman, M.L., 2001. Learning to select branching rules in the DPLL procedure for satisfiability. Electron. Notes Discrete Math. 9, 344–359.
- Legat, C., Vogel-Heuser, B., 2017. A configurable partial-order planning approach for field level operation strategies of PLC-based industry 4.0 automated manufacturing systems. Eng. Appl. Artif. Intell. 66, 128–144.
- Li, Z., Chen, Q., Koltun, V., 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In: Advances in Neural Information Processing Systems. pp. 539–548.
- Li, H., Liang, Y., Zhang, N., Guo, J., Xu, D., Li, Z., 2016. Improving degreebased variable ordering heuristics for solving constraint satisfaction problems. J. Heuristics 22 (2), 125–145.
- Liberatore, P., 2000. On the complexity of choosing the branching literal in DPLL. Artificial Intelligence 116 (1-2), 315-326.
- Ma, C., Liu, Z., Cao, Z., Song, W., Zhang, J., Zeng, W., 2020. Cost-sensitive deep forest for price prediction. Pattern Recognit. 107, 107499.
- Mackworth, A.K., Freuder, E.C., 1993. The complexity of constraint satisfaction revisited. Artificial Intelligence 59 (1–2), 57–62.
- Mao, H., Schwarzkopf, M., Venkatakrishnan, S.B., Meng, Z., Alizadeh, M., 2019. Learning scheduling algorithms for data processing clusters. In: Proceedings of the ACM Special Interest Group on Data Communication. pp. 270–288.
- Pardo, F., Tavakoli, A., Levdik, V., Kormushev, P., 2018. Time Limits in Reinforcement Learning. In: International Conference on Machine Learning. pp. 4042–4051.
- Petit, T., Trapp, A.C., 2019. Enriching solutions to combinatorial problems via solution engineering. INFORMS J. Comput. 31 (3), 429-444.
- Popescu, A., Polat-Erdeniz, S., Felfernig, A., Uta, M., Atas, M., Le, V.-M., Pilsl, K., Enzelsberger, M., Tran, T.N.T., 2021. An overview of machine learning techniques in constraint solving. J. Intell. Inf. Syst. 1–28.
- Refalo, P., 2004. Impact-based search strategies for constraint programming. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 557–571.

Rossi, F., Van Beek, P., Walsh, T., 2006. Handbook of Constraint Programming. Elsevier.

- Salido, M.A., Garrido, A., Barták, R., 2008. Introduction: Special issue on constraint satisfaction techniques for planning and scheduling problems. Eng. Appl. Artif. Intell. 21 (5), 679–682.
- Samulowitz, H., Memisevic, R., 2007. Learning to solve QBF. In: Twenty-Second AAAI Conference on Artificial Intelligence. pp. 255–260.
- Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L., 2019. Learning a SAT solver from single-bit supervision. In: International Conference on Learning Representations.
- Song, W., Kang, D., Zhang, J., Cao, Z., Xi, H., 2019. A sampling approach for proactive project scheduling under generalized time-dependent workability uncertainty. J. Artificial Intelligence Res. 64, 385–427.
- Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J., 2014. The minizinc challenge 2008–2013. AI Mag. 35 (2), 55–60.
- Wu, Y., Song, W., Cao, Z., Zhang, J., Lim, A., 2021. Learning improvement heuristics for solving routing problems. IEEE Trans. Neural Netw. Learn. Syst. 1–13. http: //dx.doi.org/10.1109/TNNLS.2021.3068828.
- Xin, L., Song, W., Cao, Z., Zhang, J., 2021a. Multi-decoder attention model with embedding glimpse for solving vehicle routing problems. In: Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence.
- Xin, L., Song, W., Cao, Z., Zhang, J., 2021b. Step-wise deep learning models for solving routing problems. IEEE Trans. Ind. Inform. 17 (7), 4861–4871. http://dx.doi.org/ 10.1109/TII.2020.3031409.
- Xu, K., Boussemart, F., Hemery, F., Lecoutre, C., 2007. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence 171 (8–9), 514–534.
- Xu, K., Hu, W., Leskovec, J., Jegelka, S., 2019. How powerful are graph neural networks? In: International Conference on Learning Representations.
- Xu, H., Koenig, S., Kumar, T.S., 2018. Towards effective deep learning for constraint satisfaction problems. In: International Conference on Principles and Practice of Constraint Programming. Springer, pp. 588–597.
- Xu, Y., Stern, D., Samulowitz, H., 2009. Learning adaptation to solve constraint satisfaction problems. Proc. Learn. Intell. Optim. (LION).
- Zarpellon, G., Jo, J., Lodi, A., Bengio, Y., 2021. Parameterizing branch-and-bound search trees to learn branching policies. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35, no. 5. pp. 3931–3939.
- Zhang, C., Song, W., Cao, Z., Zhang, J., 2020. Learning to dispatch for job shop scheduling via deep reinforcement learning. In: Advances in Neural Information Processing Systems.