Submitted to *INFORMS Journal on Computing*
manuscript MS-0001-1922.65

Improving Variable Orderings of Approximate Decision Diagrams using Reinforcement Learning

Quentin Cappart
Ecole Polytechnique de Montréal, Montreal, Canada, quentin.cappart@polymtl.ca
ServiceNow Research (formerly Element AI), Montreal, Canada

David Bergman
University of Connecticut, Stamford, CT 06901, USA, david.bergman@uconn.edu

Louis-Martin Rousseau
Ecole Polytechnique de Montréal, Montreal, Canada, louis-martin.rousseau@polymtl.ca

Isabeau Prémont-Schwarz
ServiceNow Research (formerly Element AI), Montreal, Canada
isabeau@cryptolab.net

Augustin Parjadis
Ecole Polytechnique de Montréal, Montreal, Canada, augustin.parjadis-de-lariviere@polymtl.ca

Prescriptive analytics provides organizations with scalable solutions for large-scale, automated decision-making. At the core of prescriptive analytics methodology is optimization, a field devoted to the study of algorithms that solve complex decision-making problems. Optimization algorithms rely heavily on generic methods for identifying tight bounds, which provide both solutions to problems and optimality guarantees. In the last decade, decision diagrams (DDs) have demonstrated significant advantages in obtaining bounds compared with the standard linear relaxation commonly used by commercial solvers. However, it is well-known that the quality of the bounds achieved by DDs is reliant on the ordering of variables chosen for the construction. Finding an ordering that optimizes standard metrics is an NP-hard problem. This paper studies how machine learning, specifically deep reinforcement learning (DRL), can be used to improve bounds provided by DDs, in particular through learning a good variable ordering. The introduced DRL models improve primal and dual bounds, even over standard linear programming relaxations, and are integrated in a full-fledged branch-and-bound algorithm. This paper therefore provides a novel mechanism for utilizing machine learning to tighten bounds, adding to recent research on using machine learning to obtain high-quality heuristic solutions and, for the first time, using machine learning to improve relaxation bounds through a generic bounding method. We apply the methods on a classic optimization problem, the maximum independent set, and demonstrate through computational testing that optimization bounds can be significantly improved through DRL. We provide the code to replicate the results obtained on the maximum independent set.

Key words: optimization bounds; decision diagrams; deep reinforcement learning

1. Introduction

Relaxation bounds, and mechanisms by which those bounds can be improved, are a critical component of scalable generic algorithms for solving combinatorial optimization problems. A noteworthy example is the *branch-and-bound* algorithm, which is at the core of state-of-the-art *mixed integer programming* (MIP) solvers such as CPLEX, Gurobi, or SCIP. Within this paradigm, a combinatorial optimization problem is defined by a linear formulation, which typically characterizes a space of exponential size. MIP solvers then reduce the size of the search space using mechanisms for quickly determining and improving upon objective function bounds. Assuming a maximization problem, high-quality feasible solutions, referred to as *primal solutions*, provide lower bounds whereas relaxation bounds, or *dual bounds*, obtained through a linear relaxation of the problem, provide upper bounds on the optimal value. These bounds are utilized to guide search and prune nodes for branch-and-bound search, which is required when solving an NP-hard problem. There is a weatlh of literature on the use of generic algorithms to identify primal solutions (Berthold 2006, Sadykov et al. 2019). Additionally, finding relaxation bounds and improving them is a prominent research focus in combinatorial optimization (Gamrath et al. 2015, Farràs et al. 2018). Improvements to bounds directly results in more efficient algorithms, which are required for real-world applications of prescriptive analytics.

As machine learning tools gain increasing popularity in combinatorial optimization, it is worthwhile to analyze whether machine learning can be used for the tedious task of improving optimization bounds. Finding a way to utilize the power of machine learning for this task may be a key to unlocking significant performance improvements in optimization solvers. One of the current challenges in improving optimization bounds is that the bound provided by the standard linear relaxation is *inflexible*. In other words, the algorithm used to compute the relaxation has no effect on the quality of the bound, which opens the possibility of integrating machine learning methods for improving bounds.

Decision diagrams (DDs) (Lee 1959, Bryant 1986), a recent tool in optimization (Andersen et al. 2007, Bergman et al. 2011, 2013, 2016a,b), provide a particularly well-suited framework for exploiting machine learning to tighten bounds of optimization problems that can be modeled through a *recursive formulation*. A DD can be used to either under-approximate or over-approximate the set of solutions to an optimization problem, meaning it can deliver both heuristic solutions and relaxation bounds. The advantage of *approximate* DDs over other bounding mechanisms is that

DDs are *flexible*: decisions made during their construction directly affect the bound they deliver. However, the quality of the bounds is known to be strongly related to the variable ordering considered during the construction of the DD (Bergman et al. 2012). It has been shown that finding the ordering yielding the tightest bound is NP-hard and is challenging even to model. Thus, designing methods for finding a good ordering is a major topic in the community and remains a great challenge. Delegating this task to a machine learning method, such as a reinforcement learning agent, is a possible solution.

The contribution of this paper is a learning-based approach to obtain a better bounding mechanism for combinatorial optimization problems that can be encoded by a DD. This is done by training an agent with reinforcement learning in order to achieve a variable ordering that tightens the bounds. The framework we introduce is effective in obtaining primal bounds, adding to the recent literature on using ML to find high-quality heuristic solutions, which focuses on a particular problem or a restricted category of problems. We propose a generic method suitable for any discrete optimization problem that can be expressed using a recursive formulation. Additionally, the tradeoff between the quality of primal solutions and their computation time can be easily controlled by adjusting the width of the approximate DDs. Furthermore, our approach extends our related conference paper (Cappart et al. 2019), which investigated how a reinforcement learning environment can be inferred from a DD construction. We now propose an end-to-end generic framework for learning primal solutions and dual bounds simultaneously. We provide detailed explanations of how the learning environment and the DD construction can be inferred from the same recursive formulation of a discrete optimization problem. Finally, as an extension of our related conference paper, we integrate the bounds in a DD-based branch-and-bound algorithm (Parjadis et al. 2021).

Experiments are mainly carried out on the *maximum independent set problem* and highlight that the bounds obtained are significantly better than those obtained by other heuristics reported in the literature and the linear relaxation. We further propose an application of the bounds inside a complete branch-and-bound algorithm. Results indicate that the proposed approach can achieve performance gains over CPLEX. Three other case studies (*maximum cut problem*, *set covering problem*, and *0-1 knapsack problem*) are also explored in the online supplement.

This paper is structured as follows. The next section introduces related work on DDs, reinforcement learning, and on the application of machine learning in combinatorial optimization. The relevant technical background is provided in Section 3. Section 4 contains the core contribution of the paper. It presents the generic process which relies on three main steps: (1) the definition of the environment, (2) the learning algorithm, and (3) the construction of the solution. The method is then tested on synthetic instances for the maximum independent set problem in Section 5.

2. Literature Review

Decision diagrams are a decades-old data structure that was initially introduced to represent switching circuits (Lee 1959) and for formal verification (Bryant 1986). They have been utilized more recently in the field of combinatorial optimization for encoding the feasible solutions of a problem in a structured way. They can be used, for instance, for postoptimality analysis in integer programming models (Hadzic and Hooker 2006), for representing the domain store in constraint programming (Andersen et al. 2007), for the implementation of global constraints (Verhaeghe et al. 2018), or even as a general-purpose solver (Bergman et al. 2016b). Another application, and the one considered in this work, is to provide upper and lower bounds for discrete optimization problems (Bergman et al. 2011, 2013).

With supervised and unsupervised learning, *reinforcement learning* (RL) (Sutton and Barto 2018) is one major area of machine learning. It focuses on how an agent can learn from its interactions with an environment in order to efficiently accomplish specific actions. The agent moves from state to state by performing a sequence of actions, each of which gives a specific reward or penalty. The behavior of an agent is characterized by a policy that dictates a certain action for each state the agent encounters. The goal is to learn a policy that maximizes the sum of rewards (resp. minimizes the sum of penalties) of each action performed by the agent. One difficulty that arises in this framework is the state-space explosion problem. Traditional RL algorithms suffer from a lack of scalability and are limited to low-dimensional problems. The main issue is that some states are never considered during the learning process when large state spaces are involved. With its resurgence, deep learning (LeCun et al. 2015) has provided new tools to overcome this problem. The fundamental idea is to use a deep neural network as a function approximation to generalize knowledge from visited to unknown states. This offers RL an opportunity for better approximations and enables scalability, to tackle problems that were previously intractable. Prominent examples are the superhuman performances obtained for the game of Go (Silver et al. 2016), Atari 2600 (Mnih et al. 2015) and other games (Silver et al. 2017). A formidable feature of recent reinforcement learning agents, such as AlphaZero, is their ability to self-learn from scratch. Starting from a random agent, and given no domain knowledge except the environment, they are able to achieve superhuman performances. The combination of RL with a deep network is commonly referred to as *deep reinforcement learning* (DRL).

Promising performances by RL and DRL algorithms have driven researchers to investigate the application of DRL to finding approximate solutions to NP-hard combinatorial optimization problems. Most work focuses on the classic *traveling salesman problem* (Bello et al. 2016, Deudon et al. 2018, Kool et al. 2019), with the noteworthy exception of Khalil et al. (2017) which tackles four NP-hard problems with a graph structure. They use a deep neural network in order to embed

the vertex of the graphs into features (Dai et al. 2016), while taking into account the structure of the graph. The competitive and increasingly better results of these works show the promise of this approach for finding solutions to NP-hard problems. Recently, Cappart et al. (2019) and Parjadis et al. (2021) advanced these efforts for generating dual bounds and using them inside a branch-and-bound algorithm. Our contribution is positioned as an extension of these last works.

Finally, let us highlight that learning bounds is not the only generic approach to improve branch-and-bound algorithms. For instance, He et al. (2014), Gasse et al. (2019), Cappart et al. (2021b) learn how to make good branching decisions, Tang et al. (2020) learns cutting planes, and Hutter et al. (2011) is dedicated to automatic algorithm configuration. We refer to three surveys (Lodi and Zarpellon 2017, Bengio et al. 2018, Cappart et al. 2021a) for more information on other modern learning-based methods for combinatorial optimization.

3. Technical Background

This section introduces the main concepts related to both decision diagrams and reinforcement learning relevant to our paper.

3.1. Decision Diagrams

In the context of this work, and for a binary optimization problem

$$\{\max f(x) : x \in \mathcal{X} \subseteq \mathbb{B}^n\}, \tag{$\mathcal{O}$}$$

a decision diagram is a graphical representation of a collection of solutions to ($\mathcal{O}$) through paths in a layered, arc-weighted, arc-labeled, and rooted digraph. The number of layers (of nodes) is one more than the number of variables in the problem, and each arc connects nodes in consecutive layers. Every layer except the last, which is assumed to contain a single *terminal node* $\mathbf{t}$, is associated with a unique variable in the optimization problem, and the length of a path, calculated as the sum of the weights of the arcs in the path, represents (either exactly or approximately) the objective function value of the solution it corresponds to. The solution a path corresponds to is dictated by the labels of the arc on the path, which for a binary optimization problem can be 0 or 1. The arcs specify if the variable corresponding to the layer of its originating node is set to 0 or 1, thereby dictating a vector in $\mathbb{B}^n$. Note that there is a natural extension to problems with discrete variables. In this case, there is an arc per value in the variable domain.

A DD can be viewed as the *state-transition* diagram of the path trajectories of a *recursive model* (RM) for ($\mathcal{O}$). Formally, an RM M for an optimization problem $\mathcal{O}$ is a tuple $\langle \mathsf{S}, \mathsf{T}, \mathsf{C} \rangle$, where

- S is the *state space*, containing three special states: the *root state* s^0, the *infeasible state* $\mathsf{s}^\emptyset$, and the *terminal state* s^t;

- $\mathsf{T} : \mathsf{S} \times X \times \{0,1\} \to \mathsf{S}$ is the *transition function* which maps states, variables, and *actions* (0 or 1) to states;
- $\mathsf{C} : \mathsf{S} \times X \times \{0,1\} \to \mathbb{R}$ is the *cost transition function*, which associates a cost with transitioning from state s by taking action $\mathsf{a} \in \{0,1\}$ on variable $x \in X$, typically assumed to be defined only for s, x, and a for which $\mathsf{T}(\mathsf{s}, x, \mathsf{a}) \neq \mathsf{s}^{\emptyset}$.

We think of the state s as containing information regarding which actions are feasible for the remaining variables, thereby also recording which variables are already considered, so that starting from the root state s^0, the recursive application of T will never take as argument any variable twice. We also assume that n applications of T to a state, i.e., the recursive application of actions to each variable, always concludes at $\mathsf{s}^{\emptyset}$ or $\mathsf{s}^{\mathbf{t}}$. This indicates where the collection of actions to variables, i.e., the assignment of 0 or 1 values to each variable, is feasible ($\mathsf{s}^{\mathbf{t}}$) or infeasible ($\mathsf{s}^{\emptyset}$). It is often assumed that for any variable $x_{(j)}$ and any action a, $\mathsf{T}(\mathsf{s}^{\emptyset}, x_{(j)}, \mathsf{a}) = \mathsf{s}^{\emptyset}$, so that the infeasible state $\mathsf{s}^{\emptyset}$ is absorbing. M is a valid RM for $\mathcal{O}$ if the following conditions are satisfied:

1. $\forall \mathsf{s} \in \mathsf{S}$, there exists an action $\mathsf{a} \in \{0,1\}$ such that $\mathsf{T}\left(\cdots \mathsf{T}\left(\mathsf{s}^0, x_{\sigma(1)}, \mathsf{a}^0\right), x_{\sigma(n)}, \mathsf{a}^n\right) = \mathsf{s}^{\mathbf{t}}$ for any ordering $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ of the variables;
2. $\forall x \in X$, the accumulated sum of costs through a path equals the objective cost of the function. It is shown in Equation (1), and it holds again for any ordering of the variables.

$$\mathsf{C}^{\mathsf{acc}}(x) = \mathsf{C}\left(\mathsf{s}^0, x_{\sigma(1)}, \mathsf{a}^0\right) + \cdots + \mathsf{C}\left(\cdots \mathsf{T}\left(\mathsf{s}^0, x_{\sigma(1)}, \mathsf{a}^0\right), x_{\sigma(n)}, \mathsf{a}^n\right) = f(x) \tag{1}$$

Given a valid RM M for $\mathcal{O}$, solving $\mathcal{O}$ amounts to finding actions for each variable for which $\mathsf{T}\left(\cdots \mathsf{T}\left(\mathsf{s}^0, x_{\sigma(1)}, \mathsf{a}^0\right), x_{\sigma(n)}, \mathsf{a}^n\right) = \mathsf{s}^{\mathbf{t}}$ that maximize $\mathsf{C}^{\mathsf{acc}}(x)$. DDs provide a mechanism by which the problem of finding the optimal actions can be solved.

A DD can be built by the following procedure. Starting from $S_1 = \{\mathsf{s}^0\}$, select any variable and designate it as $x_{(1)}$. Additionally, maintain an auxiliary digraph that will become the DD, initialized as a single node called the *root node*, denoted by $\mathbf{r}$. Each node u will belong to a *layer* $\mathbf{l}(u) \in \{1, \ldots, n+1\}$, with $\mathbf{L}_j := \{u : \mathbf{l}(u) = j\}$ for $j = 1, \ldots, n+1$, and correspond to some state $\mathsf{s}(u) \in \mathsf{S}$. Each layer j, for $j = 1, \ldots, n$, will be assigned a variable $x_{\sigma(j)}$. The first layer consists of a single node, $\mathbf{L}_1 = \{\mathbf{r}\}$, with $\mathsf{s}(\mathbf{r}) = \mathsf{s}^0$ and $\mathbf{l}(r) = 1$.

The layers of the DD are built iteratively. Having constructed layer $\mathbf{L}_j$, we build layer $\mathbf{L}_{j+1}$ and all arcs that connect nodes between these successive layers. Each arc a is directed from $\mathbf{u}^-(a)$ to $\mathbf{u}^+(a)$, where it is assumed that $\mathbf{l}(\mathbf{u}^+(a)) = \mathbf{l}(\mathbf{u}^-(a)) + 1$. The arcs have a *domain* $\mathbf{d}(a) \in \{0,1\}$ when solving a binary optimization problem. The *arc cost* $c(a)$ is defined by its domain, as well as the state of its tail and its layer's associated variable, $\mathbf{c}(a) = \mathsf{C}\left(\mathsf{s}(\mathbf{u}^-(a)), x_{(\mathbf{l}(\mathbf{u}^-(a)))}, \mathbf{d}(a)\right)$, which is the cost of taking action $\mathbf{d}(a)$ for variable $x_{(\mathbf{l}(\mathbf{u}^-(a)))}$ transitioning from state $\mathsf{s}(\mathbf{u}^-(a))$.

The first step in creating $\mathbf{L}_{j+1}$ is to select a variable $x_{(j)}$ that has not been used for the previous layer. This is the major consideration of this paper, to be addressed and discussed in subsequent sections. Once $x_{(j)}$ is selected, each node $u \in \mathbf{L}_j$ is processed by calculating $\mathsf{s}' := \mathsf{T}(\mathsf{s}(u), x_{\sigma(j)}, \mathsf{a})$, for a equals 0 and 1. If $\mathsf{s}' \neq \mathsf{s}^{\emptyset}$, we check whether there exists a node u' such that $\mathsf{s}'(u') = \mathbf{s}(u)$, for any u' in the current set of nodes in $\mathbf{L}_{j+1}$. If such a node u' exists, arc a' is added with $\mathbf{u}^-(a') = u$ and $\mathbf{u}^+(a') = u'$. Otherwise, we create a new node u', add it to $\mathbf{L}_{j+1}$, and add the arc a'. We set $\mathbf{d}(a) = \mathsf{a}$. Note that when processing the penultimate layer, i.e., $j = n$, when creating layer $\mathbf{L}_{n+1}$, only state $\mathsf{s}^{\mathbf{t}}$ is possible and so $|\mathsf{L}_{n+1}| \leq 1$.

Equipped with the DD for M that encodes the maximization problem $\mathcal{O}$, finding a sequence of optimal actions reduces to finding the longest path from $\mathbf{r}$ to $\mathbf{t}$ using $\mathbf{c}(a)$ as the length of each arc. The correspondence can be seen by viewing an arc a' as setting the variable $x_{\mathbf{l}(\mathbf{u}^-(a'))} = \mathbf{d}(a')$ so that any $\mathbf{r}$ to $\mathbf{t}$ path leads to the assignment of a value to each variable. If M is a valid RM, there is a one-to-one correspondence of the collection of $\mathbf{r}$ to $\mathbf{t}$ paths, which we denote by $\mathcal{P}$, and solutions to $\mathcal{O}$, with the length of each path equal to the value of the solution. Letting $\mathcal{X}(\mathcal{P})$ be the collection of solutions dictated by paths, we have that $\mathcal{X}(\mathcal{P}) = \mathcal{X}$. For any $\mathbf{r}$ to $\mathbf{t}$ path $p = (a_1, \ldots, a_n)$, let $x(p)$ be the solution it corresponds to (i.e., $x(p)_{\sigma(j)} = \mathbf{d}(a_j)$) and $\mathbf{c}(p)$ its length (i.e., $\mathbf{c}(p) = \sum_{j=1}^{n} c(a_j)$).

An optimal collection of actions, which corresponds to an optimal solution to $\mathcal{O}$, can be found in linear time in the size of the DD, because it is acyclic. The challenge faced in this process is the well-known *curse of dimensionality*, which may render an exponentially-sized DD. A solution to this exponential growth is to build *limited-width approximate* DDs that provide either primal solutions (in the case of *restricted* DDs) or dual bounds (in the case of *relaxed* DDs).

A DD of width W is a DD for which $|\mathbf{L}_j| \leq W$ for $j = 1, \ldots, n+1$. A *restricted DD* is a DD for which a subset of the solutions to $\mathcal{O}$ are represented, i.e., $\mathcal{X}(\mathcal{P}) \subset \mathcal{X}$. A restricted DD of width W is easily constructed by dropping a collection of nodes from $\mathcal{L}_j$ after its construction, keeping only a subset of those nodes of size W to expand. Since $\mathcal{X}(\mathcal{P}) \subset \mathcal{X}$, the longest path in this DD corresponds to a feasible solution to $\mathcal{O}$, thereby providing (in the case of maximization) lower bounds on the optimal value of $\mathcal{O}$.

A *relaxed DD* is a DD satisfying the following two properties: (1) $\mathcal{X}(\mathcal{P}) \supseteq \mathcal{X}$, and (2) $\forall p \in \mathcal{P}, \mathbf{c}(p) \geq f(x(p))$. The length of the longest path in a relaxed DD is therefore a dual bound. The construction of a relaxed DD of width W requires the definition of a *relaxation operation* $\otimes$, which takes two states s_1 and s_2 and merges them in manner in which a superset of the available paths will be possible and/or the lengths of the paths increase. This is similar to aggregation techniques in state-space relaxations (Christofides et al. 1981) and has been investigated for a variety of combinatorial optimization problems (Bergman and Cire 2018). In order to illustrate the concepts introduced throughout this paper, we refer to an example, the *maximum independent set problem*, for which the impact of variable ordering has been thoroughly studied (Bergman et al. 2012).

Definition 1 (maximum independent set problem). Let $G(V, E)$ be a simple, undirected graph. An independent set of G is a subset of vertices $I \subseteq V$ such that there are no two vertices in I that are connected by an edge of E. The *maximum independent set problem* (MISP) consists in finding an independent set with the largest cardinality. A standard binary optimization model of the MISP is as follows, letting $V = \{1, \dots, n\}$:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} x_j \\ \text{s.t.} \quad & x_i + x_j \leq 1, \quad \forall \{i,j\} \in E \\ & x_j \in \{0,1\} \quad \forall j \in V \end{aligned}$$

Example 1 (Representing MISP decision diagrams). A possible recursive formulation for this problem is found in Bergman et al. (2012), and is defined as follows:

1. $\mathsf{S} : 2^V$, with $\mathsf{s}^0 = V$
2. $\mathsf{T}\left(\mathsf{s}, x_{\sigma(j)}, \mathsf{a}\right) = \begin{cases} \mathsf{s} \backslash \{\sigma(j)\}, & \text{if } \mathsf{a} = 0, \\ \mathsf{s} \backslash \{N[\sigma(j)]\}, & \text{if } \mathsf{a} = 1, \text{and } \sigma(j) \in \mathsf{s}, \\ \mathsf{s}^{\emptyset}, & \text{otherwise,} \end{cases}$
3. $\mathsf{C}\left(\mathsf{s}, x_{\sigma(j)}, \mathsf{a}\right) = \begin{cases} 0, & \text{if } \mathsf{a} = 0, \\ 1, & \text{if } \mathsf{a} = 1, \text{and } \sigma(j) \in \mathsf{s}, \\ \text{undefined}, & \text{otherwise,} \end{cases}$

where $N[j] = \{j\} \cup \{i : \{i,j\} \in E\}$ is the *closed neighborhood* of j. Additionally, the merging operator $\otimes$ can be defined as the standard set union operator (Bergman et al. 2013). Consider the top graph in Figure 1. The largest independent set has cardinality 2 (four optimal solutions: $\{1,4\}, \{1,5\}, \{2,5\}, \{3,5\}$). This figure depicts two DDs for the MISP on this graph, obtained using two different choices of variable orderings. A solid line corresponds to an arc that sets a variable equal to 1, and a dashed arc corresponds to an arc that sets a variable equal to 0. Both contain 10 paths corresponding to the 10 feasible solutions, and, if we associate an arc-cost of 1 to each solid arc and an arc-cost of 0 to each dashed arc, the longest path in both is 2 (optimal value).

This example illuminates how sensitive DD construction is to variable ordering. Both DDs represent the same set of solutions, but the one on the right contains 3 fewer nodes and 7 fewer arcs. Figure 2 depicts a restricted DD (left) and a relaxed DD (right) for the same MISP instance using the variable ordering for the exact DD in Figure 1 depicted on the left. The restricted DD contains 5 paths, each corresponding to a feasible solution. The longest path is still of length 2, the optimal value, but in general will yield a lower bound. The relaxed DD contains 12 paths, 2 of which are infeasible. The longest path has length 3 (solid-solid-solid-dashed-dashed path) that corresponds to the solution $(1, 0, 0, 1, 1)$. Although infeasible, this solution provides a dual bound of 3.

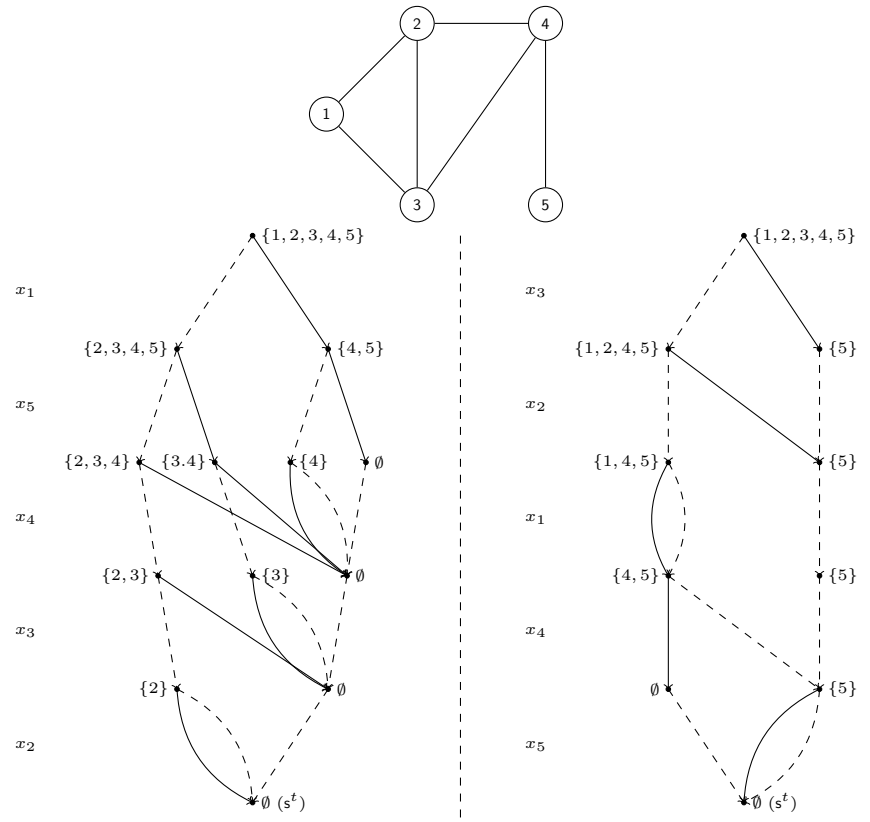


Figure 1 **Two exact DDs for a specific MISP instance, with $\sigma = (1, 5, 4, 3, 2)$ (left) and $\sigma = (3, 2, 1, 4, 5)$ (right). The DDs are built using the recursive formulation proposed in Example 1 and two different variable orderings. Dashed lines correspond to a 0 variable assignment, and solid lines to 1.**

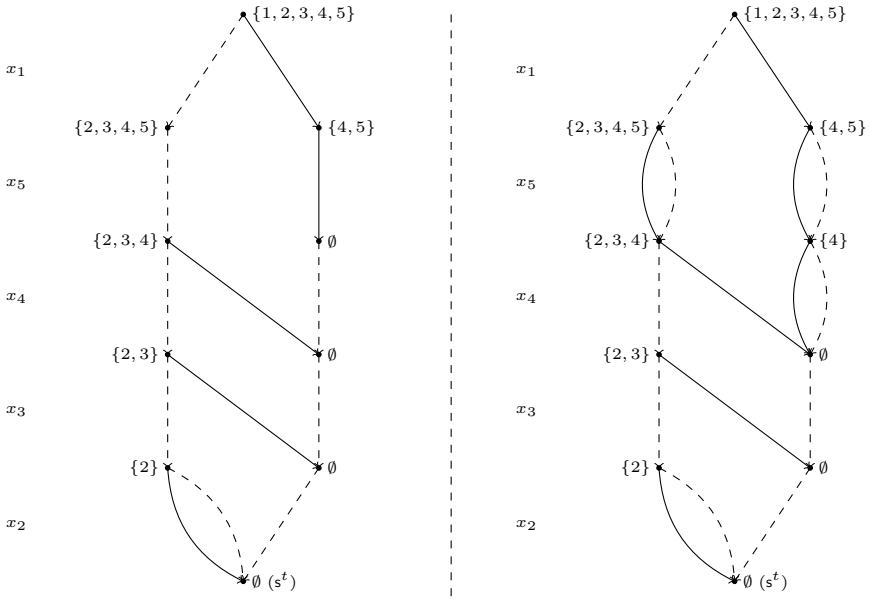


Figure 2 **A restricted (left) and relaxed (right) DD obtained from the left DD in Figure 1 ($\sigma = (1, 5, 4, 3, 2)$).**

3.2. Reinforcement Learning

Let $\langle S, A, T, R\rangle$ be a tuple representing a deterministic agent-environment pair 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 the transition function leading the agent from one state to another one given the action taken, and $R : S \times A \to \mathbb{R}$ is the reward function of taking an action from a specific state. This model is consistent with the definition of a recursive model as defined in the previous section. The behavior of an agent is defined by a policy $\pi : S \to A$ describing the action to be done given a specific state. The goal of an agent is to learn a policy maximizing the accumulated sum of rewards (eventually discounted) during its lifetime, defined by a sequence of states $s_t \in S$ with $t \in [1, \ldots, \Theta]$. Such a sequence is called an *episode*, where s_Θ is the terminal state. The expected return after time step t is denoted by G_t and is defined in Equation (2), where $\gamma \in [0, 1]$ is a discounting factor used for parametrizing the weight of future rewards. When $\gamma = 1$, all rewards have the same importance.

$$G_t = \sum_{k=t+1}^{\Theta} \gamma^{k-t-1} R(s_k, a_k) \tag{2}$$

For a deterministic environment, the quality of taking an action a from a state s under a policy π is defined by the action-value function $Q^\pi(s, a) = G_t$. The challenge is to find a policy that maximizes the expected return: $\pi^\star = \arg\max_\pi Q^\pi(s, a)\ \forall s \in S, \forall a \in A$. In practice, $\pi^\star$ is computed from an initial policy by two nested operations: (1) the policy evaluation, which makes the action-value function consistent with the current policy, and (2) the policy iteration, which greedily improves the current policy. However, in most problems, the optimal policy, or even an optimal action-value function, cannot be computed in a reasonable amount of time. A method based on approximation, such as Q-learning (Watkins and Dayan 1992), is therefore required. Instead of computing the optimal action-value function, Q-learning approximates the function by iteratively updating the current estimate after each action. The update function is defined in Equation (3), where $\alpha \in (0, 1]$ is the learning rate.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(R(s_t, a_t) + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)\Big) \tag{3}$$

Another issue that arises for large problems is that almost every state encountered may never have been seen during previous updates, thus necessitating a method capable of utilizing prior knowledge to generalize for different states that share similarities. Among such methods, *neural fitted Q-learning* (Riedmiller 2005) uses a neural network for approximating the action-value function. This provides an estimator $\widehat{Q}(s, a, \mathbf{w}) \approx Q(s, a)$, where $\mathbf{w}$ is a weight vector that is learned. Stochastic gradient descent, or another optimizer coupled with back-propagation (Rumelhart et al. 1986), is then used for updating $\mathbf{w}$ and aims to minimize the squared loss between the current

Q-value and the new value assigned using Q-learning. This approach is shown in Equation (4) where $L(\mathbf{w})$ is the square loss as defined in Equation (5).

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{1}{2}\alpha \nabla L(\mathbf{w}) \tag{4}$$

$$L(\mathbf{w}) = \Big(R(s_t, a_t) + \gamma \max_{a \in A} \widehat{Q}(s_{t+1}, a, \mathbf{w}) - \widehat{Q}(s_t, a_t, \mathbf{w})\Big)^2 \tag{5}$$

Other reinforcement learning algorithms exist in the literature that do not aim to select actions based on their estimated Q-values. These algorithms generally use *policy gradient methods*, where, a parametrized policy selects actions without resorting to the action-value function. The function may still be used for learning the parameter of the policy but not for the action selection.

4. Generic Framework for Learning Optimization Bounds

This section describes the framework we propose for learning optimization bounds in a generic fashion. It is important to note that both technologies are based on a recursive formulation. The idea is to use this unique representation to define both the DD construction and the reinforcement learning environment. The process is divided into two phases:

1. *The learning phase*, where a model is trained to identify good variable orderings. A set of instances, either from a known dataset or randomly generated, as well as the recursive formulation, are given as input for the reinforcement learning framework. The model, implemented as a deep neural network, is then parametrized.

2. *The evaluation phase*, where the model is used to drive the variable ordering used for building the relaxed and restricted DDs of the instances we want to evaluate. The same recursive formulation is used for the DD construction. The expectation is that by using the ordering given by the learned model, tight optimization bounds will be obtained.

The complete process for this approach is illustrated in Figure 3. The cubic blocks represent the components that are problem-specific, such as the recursive formulation, or specific to a class of problems, such as the neural architecture or the relaxation operator. For instance, a graph neural network (Scarselli et al. 2008) can be used for all problems that can be represented as a graph. Once these components have been defined, the process follows these steps:

(1) A training set is built either using known instances or using a random generator. The training instances can also be generated dynamically during the training phase.

(2) The reinforcement learning framework operates the learning using the training instances as input. The output is two parametrized neural networks, one designed for the relaxed DDs and the other for the restricted DDs. The environment is directly inferred from the recursive formulation.

(3) The recursive formulation is then used for building the relaxed and restricted DDs of the instance that must be evaluated. The previously trained models are used for determining an appropriate variable ordering for both DDs.

(4) Finally, the DDs give a primal and a dual bound for the the evaluated instance. Such bounds can be used afterward in a branch-and-bound algorithm.

This framework, using reinforcement learning to drive the construction of DDs in order to improve optimization bounds, is the main contribution of this paper. It is suited for every optimization problem that can be expressed by a recursive formulation. A relaxation operator is also required when computing the dual bound. The rest of this section describes the generic reinforcement learning environment we designed and the learning algorithm we used in order to solve it.

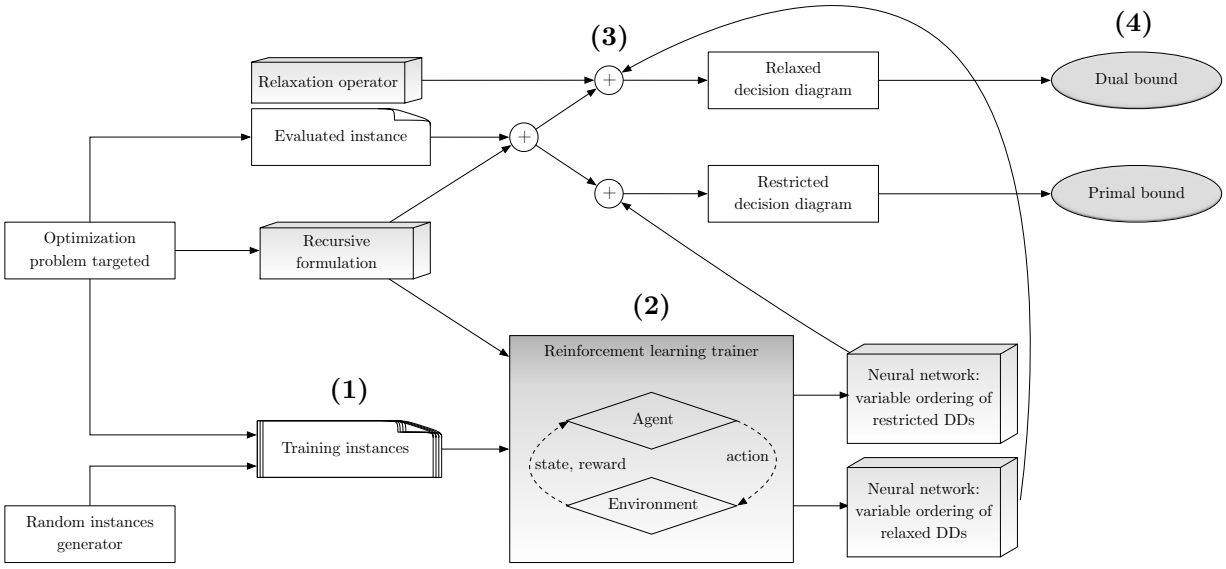


Figure 3 **Complete process of the framework: (1) a training set is built, (2) a training based on RL is conducted on generated instances in order to learn variable orderings giving tight bounds, (3) the ordering is used to compile relaxed and restricted DDs, and (4) the DDs are used to compute primal and dual bounds.**

4.1. Reinforcement Learning Environment

Let $P = \langle X, D, C, O \rangle$ be a *constrained optimization problem* (COP), where X represents the set of variables, D the set of discrete domains restricting the variables $x \in X$, C the set of constraints and O the objective function. Designing an RL model for determining the variable ordering of a DD associated with P requires defining, the tuple $\langle S, A, T, R \rangle$ to represent the system. The environment we designed is defined as follows.

- **State**: An RL state $s \in S$ is a pair $\langle s_L, s_B \rangle$ containing an ordered sequence of variables s_L and a partial DD s_B associated with variables in s_L. A state s is terminal if s_L includes all the variables of X. We emphasize that an RL state does not correspond to a DD state (i.e., a node).

- **Action**: An action is defined as the selection of a variable from X. An action a_s can be performed at the RL state s if and only if it is not yet inserted in s_B (i.e., $a_s \in X \setminus s_L$).
- **Transition function**: A transition is a function updating the current RL state according to the action performed. Let $B \oplus x$ be an operator adding the variable x into a decision diagram B, and $y :: x$ another operator appending the variable x to an arbitrary ordered sequence of variables (y); we have $T(s, a_s) = \langle s_L :: a_s, s_B \oplus a_s \rangle$.
- **Reward function**: The reward function is designed to tighten the bounds obtained with the DD. For maximization problems, upper bounds are provided by relaxed DDs and lower bounds by restricted DDs. Both cases are associated with a common reward. Let $\lceil B \rceil$ and $\lfloor B \rfloor$ indicate the current upper/lower bound obtained with the decision diagram B. Such bounds correspond to the current partial longest path of the relaxed/restricted DD from the root node to the last constructed layer. At each variable insertion in B, the difference in the longest path when adding the new layer is computed. When computing the upper bound, this difference is penalized because we want the bound to be as small as possible (Equation (6)). For the lower bound, this difference is rewarded, as we want the bound to be as large as possible (Equation (7)).

$$R^{ub}(s, a_s) = -\big(\lceil s_B \oplus a_s \rceil - \lceil s_B \rceil\big) \tag{6}$$

$$R^{lb}(s, a_s) = \big(\lfloor s_B \oplus a_s \rfloor - \lfloor s_B \rfloor\big) \tag{7}$$

When the objective is to minimize, the shortest path must be considered instead of the longest one. The upper bounds are then provided by restricted DDs and lower bounds by relaxed DDs.

4.2. Learning Algorithm

The next step is to use a reinforcement learning algorithm to find a good sequence of actions. The core of the learning algorithm we have chosen relies on *neural fitted Q-learning*, as described previously, and includes several improvements:

- *Experience replay*: Let $\langle s, a, r, s' \rangle$ be a sample representing an action done (a) at a specific state (s) with its reward (r) and the next state reached (s'), and let $\mathcal{D}$ be a sample store with a fixed memory. Each time an action is performed, $\langle s, a, r, s' \rangle$ is added in $\mathcal{D}$. Then, the optimizer updates $\mathbf{w}$ using a random sample taken from $\mathcal{D}$ (Lin 1992). However, uniformly sampling the experiences from the memory does not take into account the relative significance of the different transitions. *Prioritized experience replay* (Schaul et al. 2015) deals with this issue by replaying important transitions more frequently.
- *Multi-step learning*: Delayed rewards, where the final reward of interest is only received far in the future during an episode, can be an issue for the learning. For this reason, updating the Q-values by looking only at the next step, as in Equation (3), may be too myopic. A possible

solution is to wait more steps before updating the parameters and then compute a more accurate estimate of the future rewards. This is the basic idea behind *multi-step Q-learning*. The update function and the square loss function (Equation (5)) are thus modified as presented in Equations (8) and (9) where n is the number of steps considered.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(\sum_{i=0}^{n-1} R(s_{t+i}, a_{t+i}) + \gamma \max_{a \in A} Q(s_{t+n}, a) - Q(s_t, a_t)\Big) \tag{8}$$

$$L(\mathbf{w}) = \Big(\sum_{i=0}^{n-1} R(s_{t+i}, a_{t+i}) + \gamma \max_{a \in A} \widehat{Q}(s_{t+n}, a, \mathbf{w}) - \widehat{Q}(s_t, a_t, \mathbf{w})\Big)^2 \tag{9}$$

- *Mini-batches*: Instead of updating the $\widehat{Q}$-function using a single sample as previously explained, it is also possible to update it by considering a mini-batch of m samples from the store memory $\mathcal{D}$. As stressed by Masters and Luschi (2018), the choice of the mini-batch size can have a huge impact on the learning process. On one hand, large batches result in faster learning and leverage better support for parallelism, especially on GPUs. On the other hand, small batches can provide a better generalization. Let $L_j(\mathbf{w})$ be the squared loss related to a sample j, with N as the batch size. The gradient update, where the square loss of each sample is summed, is defined in Equation (10).

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{1}{2N}\alpha \sum_{j=1}^{N} \nabla L_j(\mathbf{w}) \tag{10}$$

- *Adaptive ϵ-greedy*: Always following a greedy policy results in a lack of exploration during learning. One solution is to introduce limited randomness in choosing an action. ϵ-greedy refers to taking a random action with probability ϵ where $\epsilon \ll 1$. Otherwise, the current policy is followed. In our case, ϵ is adaptive and decreases linearly during the learning process, resulting in focused exploration at first, followed by increased exploitation.
- *Reward scaling*: Gradient-based methods have difficulty learning when rewards are large or sparse. *Reward scaling* compresses the space of rewards into a smaller interval value near zero while still remaining sufficiently large, since tiny rewards can also lead to failed learning, as stressed by Henderson et al. (2018). We let $\rho \in \mathbb{R}$ be the scaling factor, generally defined as a power of 10, and rescale the rewards as $r_t = \rho R(s_t, a_t)$.

Other improvements, such as the integration of a target network or double Q-learning (Van Hasselt et al. 2016), are also possible. The complete algorithm is presented in Algorithm 1.

Algorithm 1: Learning Algorithm.

1 ▷ **Pre:** $\mathcal{M}$ is the training set containing COPs.
2 ▷ K is the number of iterations.
3 ▷ N is the batch size.
4 ▷ $\epsilon, \rho, \alpha, \gamma$, are parameters as defined previously.
5 ▷ π is the policy, which is randomly initialized.
6 ▷ $\mathbf{w}$ is the weight vector, which is randomly initialized.
7 $\mathcal{D} := \emptyset$ ▷ *Experience replay store*
8 **for** i **from** 1 **to** K **do**
9 $P, V := \text{randomValueFrom}(\mathcal{M})$
10 $\langle S, A, T, R \rangle := \text{initializeEnvironment}(P)$ ▷ *Reinforcement learning environment*
11 $s_1 := \langle \emptyset, \emptyset \rangle$ ▷ *Decision diagram is empty*
12 **for** t **from** 1 **to** V **do**
13 $\pi := \arg\max_{\pi} \widehat{Q}^{\pi}(s, a, \mathbf{w})$ $\forall s \in S, \forall a \in A$
14 $k := \text{randomValueFrom}([0, 1])$
15 **if** $k > \epsilon$ **then**
16 $a_t := \pi(s_t)$ ▷ *Following policy*
17 **else**
18 $a_t := \text{randomValueFrom}(A)$ ▷ *ϵ-greedy*
19 $r_t := \rho R(s_t, a_t)$ ▷ *Reward scaling*
20 $s_{t+1} := T(s_t, a_t)$
21 $\mathcal{D} := \mathcal{D} \cup \{\langle s_t, a_t, r_t, s_{t+1} \rangle\}$ ▷ *Store update*
22 **for** j **from** 1 **to** N **do**
23 $\langle s_e, a_e, r_e, s'_e \rangle := \text{randomValueFrom}(\mathcal{D})$ ▷ *Experience replay selection*
24 $L_j(\mathbf{w}) := \left(r_e + \gamma \max_{a \in A} \widehat{Q}(s'_e, a, \mathbf{w}) - \widehat{Q}(s_e, a_e, \mathbf{w})\right)^2$ ▷ *Square loss*
25 $\mathbf{w} := \mathbf{w} - \frac{1}{2N} \alpha \sum_{j=1}^{N} \nabla L_j(\mathbf{w})$ ▷ *Mini-batch*
26 update(ϵ)
27 **return w**

At each iteration, a COP (P) of V variables is randomly taken from the training set and the learning is conducted on it (line 9). Effective learning for any particular class of COPs should consider instances for that class of COP. For example, if the goal is to find objective function bounds for an instance of the maximum independent set problem, other instances from that class of problem should be used during the training. The reinforcement learning environment is then initialized (line 10). The next loop (lines 12-26) iterates until all the variables of P have been considered. According to the ϵ-greedy threshold, the next action to do is either random (line 17) or follows the current greedy policy (line 15). The resulting reward is then scaled (line 18), and the experience is stored in a buffer (line 20). As long as the batch size is not exceeded, random samples from the experience replay store are picked up and used for computing the square loss (lines 21-23). The weights of the deep network are then updated (line 24). Finally, the algorithm returns a vector of weights ($\mathbf{w}$) which is used for parametrizing the approximate action-value function $\widehat{Q}$.

4.3. Construction of the Solution

The trained model is used for building the DD related to the problem we want to solve. The process is shown in Algorithm 2. It takes as input the weight vector that was obtained in Algorithm 1 for setting the action-value function and the optimization problem we want to evaluate. The greedy policy (line 5) is followed for selecting the next variable until all of them have been considered (lines 7-9). The last RL state, corresponding an entirely built DD, is finally returned (line 10). An illustration of this process is described in Example 2.

Algorithm 2: Construction of the Decision Diagram.

1 ▷ **Pre:** P is the optimization problem we want to solve.
2 ▷ V is the number of variables in P.
3 ▷ $\mathbf{w}$ is the weight vector, which has been computed in Algorithm 1.
4 $\langle S, A, T, R\rangle := \text{initializeEnvironment}(P)$
5 $\pi := \arg\max_\pi \widehat{Q}^\pi(s, a, \mathbf{w})$ $\quad \forall s \in S, \forall a \in A$
6 $s_1 := \langle \emptyset, \emptyset \rangle$
7 **for** t **from** 1 **to** V **do**
8 $\quad a_t := \pi(s_t)$
9 $\quad s_{t+1} := T(s_t, a_t)$
10 **return** s_V

Example 2. Let us consider the example of Figure 1 (left part) of the MISP. Table on the top of Figure 4 shows examples of Q-values that can be obtained using the model trained in Algorithm 1. Following the greedy policy, the variable with the highest Q-value is always selected, which in turn is used to select the next variable for compiling the DD.

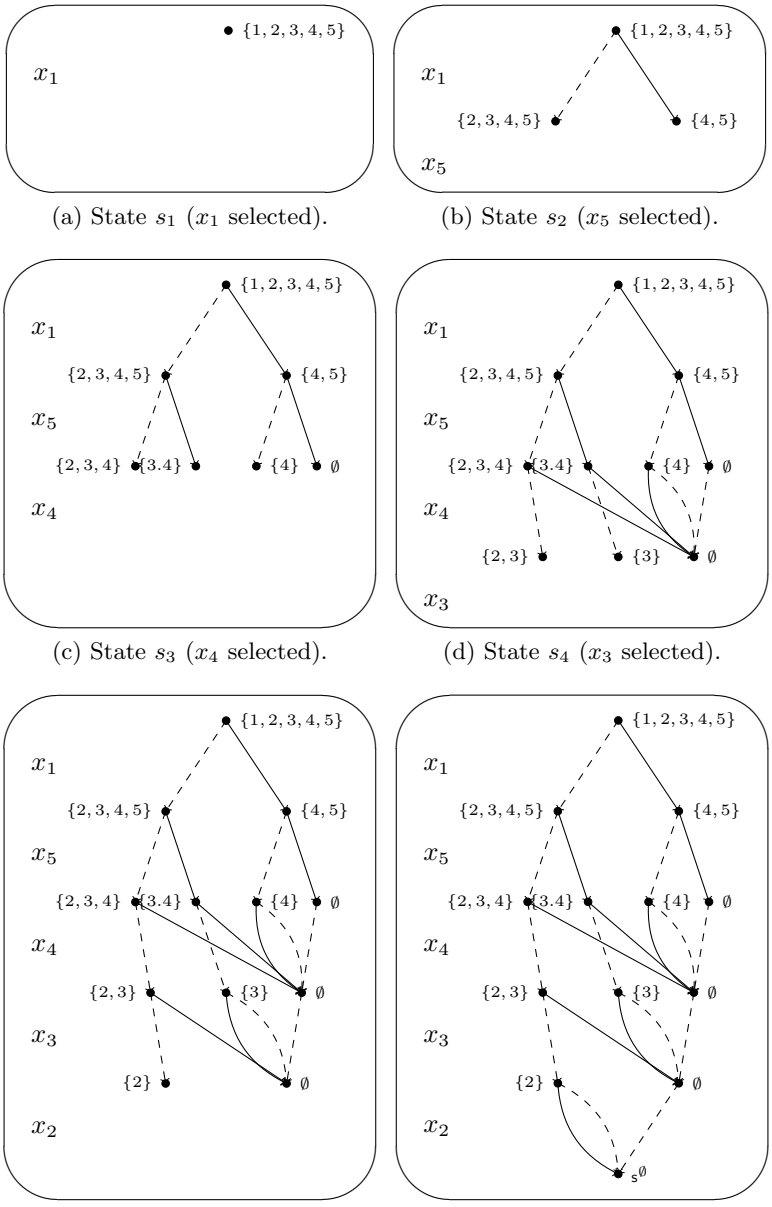
5. Case Study: Maximum Independent Set Problem

As is commonly done in the literature related to DDs, we propose to validate our approach on the MISP. This section describes the neural architecture we consider to approximate the Q-values, the experimental protocol, and finally the experiments we carried out. The implementation is available online on Github (`https://github.com/corail-research/learning-to-bound`).

5.1. Neural Network Architecture

An important design choice is the selection of a neural network architecture. It must encompass the combinatorial structure of the problem, and give as output an estimated Q-value for each variable of the problem at each stage of the DD construction. Interestingly, there is a one-to-one matching with a partial DD, and the related MISP instance partially solved. We can directly use the MISP instance as input for the neural network. A fundamental aspect of the MISP is that it can be entirely represented as a graph, which motivates the use of a graph neural network architecture (Scarselli et al. 2008). Following the work of Khalil et al. (2017), we reuse the `structure2Vec` (Dai et al. 2016) embedding. Let $G(V, E)$ be a simple, undirected graph with V the set of vertices, E the set of edges, $\mathbf{f}_v \in \mathbb{R}^{k_v}$ a vector of features attached to a vertex $v \in V$, and $\mathbf{h}_e \in \mathbb{R}^{k_e}$ a vector

		Actions				
		x_1	x_2	x_3	x_4	x_5
States	s_1	7	1	3	2	6
	s_2	-	4	2	0	5
	s_3	-	2	5	8	-
	s_4	-	0	2	-	-
	s_5	-	1	-	-	-
	s_6	-	-	-	-	-



(a) State s_1 (x_1 selected). (b) State s_2 (x_5 selected).

(c) State s_3 (x_4 selected). (d) State s_4 (x_3 selected).

(e) State s_5 (x_2 selected). (f) State s_6 (terminal state).

Figure 4 Example of an exact DD construction, step-by-step, for a MISP instance, following the policy $\pi = \arg\max_a \widehat{Q}^{\pi}(s, a, \mathbf{w})$ based on the output of Algorithm 1.

of features attached to an edge $e \in E$, where k_v is the number of features of vertex v and k_e the number of features of edge e. The objective of `structure2vec` is to compute a p-dimensional feature embedding $\mu_v \in \mathbb{R}^p$ for each node $v \in V$. Details of this specific architecture are proposed in the online supplement. We refer to the survey from Cappart et al. (2021a) for more information about the relevance of graph neural networks in combinatorial optimization.

The neural network is used to obtain an approximate Q-value for each state-action pair. Typically, an action corresponds to the choice of a vertex in the subset of non-selected vertices of the initial graph, while the state corresponds to a combinatorial structure dependent on the previously-chosen vertices. This process is illustrated in Figure 5, where μ is the message passing operation, and z_v the embedding that has been computed and which serves as input for a fully-connected neural network. Once trained, the network will be called each time a Q-value must be computed.

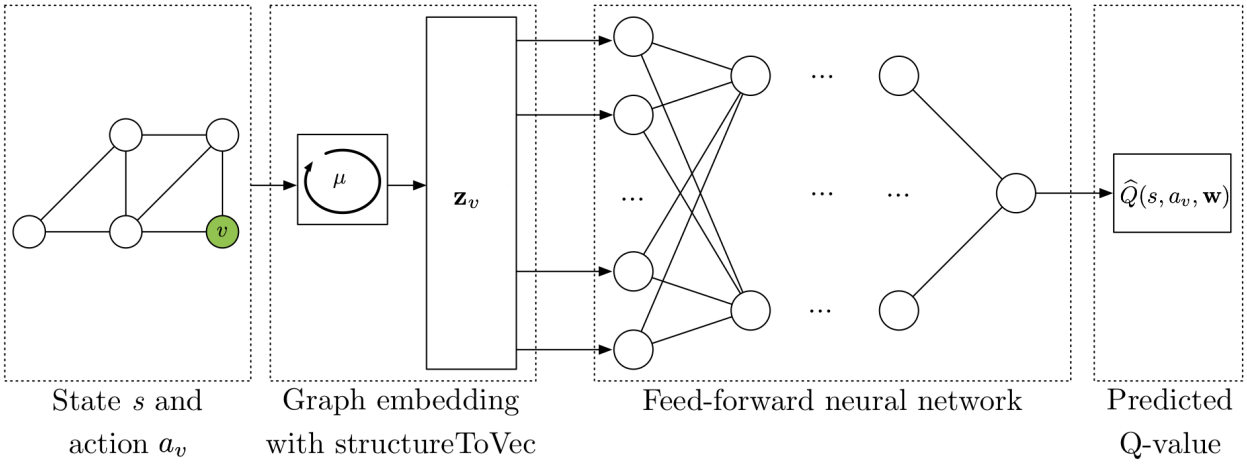


Figure 5 Approximated Q-value obtained from a state using a graph embedding and a deep neural network.

Finally, a binary feature b_v is added to each node $v \in V$. It is set to 1 if v has already been considered during the DD construction and to 0 otherwise. The features are updated after each action. No modification on the graph structure is done. No information concerning the edges is used. It is worth noting that this encoding contains all the required information to build the RL state (i.e., related partial DD). No further information from the DD is required.

5.2. Experimental Protocol

Instance generation Except for the comparisons with CPLEX, instances are generated using the model proposed by Albert and Barabási (2002). This model is commonly used to generate scale-free graphs that mimic real-world networks. They are defined by the number of nodes (n) and an attachment parameter (ν). The greater ν is, the denser the graph. Edges are added preferentially to nodes with a higher degree. The graph size is randomly sampled from a uniform distribution ($n \in \{90, \dots, 100\}$). Four models, with a specific attachment parameter, are trained ($\nu \in \{2, 4, 8, 16\}$).

Baseline Comparisons are performed against the linear relaxation bound (`LP`), which is obtained using a standard clique formulation for the MISP as described by Bergman et al. (2013), a random ordering (`RAND`; best, worst, and average bounds obtained among 100 trials), and three heuristics commonly used in the literature:

1. *Maximal Path Decomposition* (`MPD`): A *maximal path decomposition* is pre-computed and used as the ordering of the vertices (Bergman et al. 2013). This ordering bounds the width of the exact DDs by the Fibonacci numbers.

2. *Minimum Number of States* (`MIN`): Having constructed up to layer j, and hence chosen the first $j-1$ vertices, the next vertex is selected as the one appearing in the fewest number of states in the DD nodes in layer j. This heuristic aims to greedily minimize the size of the next layer.

3. *Minimum Vertex Degree* (`DEG`): The vertices are ordered in ascending order of vertex degree. The vertices with the lowest degree are inserted first.

Training and Evaluation Training time is limited to 24 hours, memory consumption to 128 GB, and one GPU (NVIDIA Tesla V100 32 GB Passive) is used. The learning is done using the Adam optimizer (Kingma and Ba 2014) with default parameters ($\beta_1 = 0,9$, $\beta_2 = 0.99$), except for the learning rate, which is dependent on the case study considered. For each configuration, the training is done using random instances that are generated dynamically during the training (i.e., a new instance is generated for each episode). The model selected is the one that gives the best average reward on a validation set composed of 100 instances randomly generated with the same generator. The training time required to get the model is dependent on the configuration considered and varies between 13 minutes and 385 minutes. The different values are summarized in the online supplement The evaluation is carried out on 100 other random graphs that are generated in the same manner as those used for the training. Performance profiles (Dolan and Moré 2002) are used to compare the learned models with other methods. This tool provides a synthetic view on how an approach performs compared to the others tested. The metric considered is the optimality gap (i.e., the relative distance between the bound and the optimal solution).

Implementation Details The model is implemented based on code from Khalil et al. (2017) for the learning part, while code from Bergman et al. (2013) is used for building the DDs of the MISP instances. Graphs are generated using the `networkX` package (Hagberg et al. 2008). The hyperparameters used are presented in the online supplement.

5.3. Results

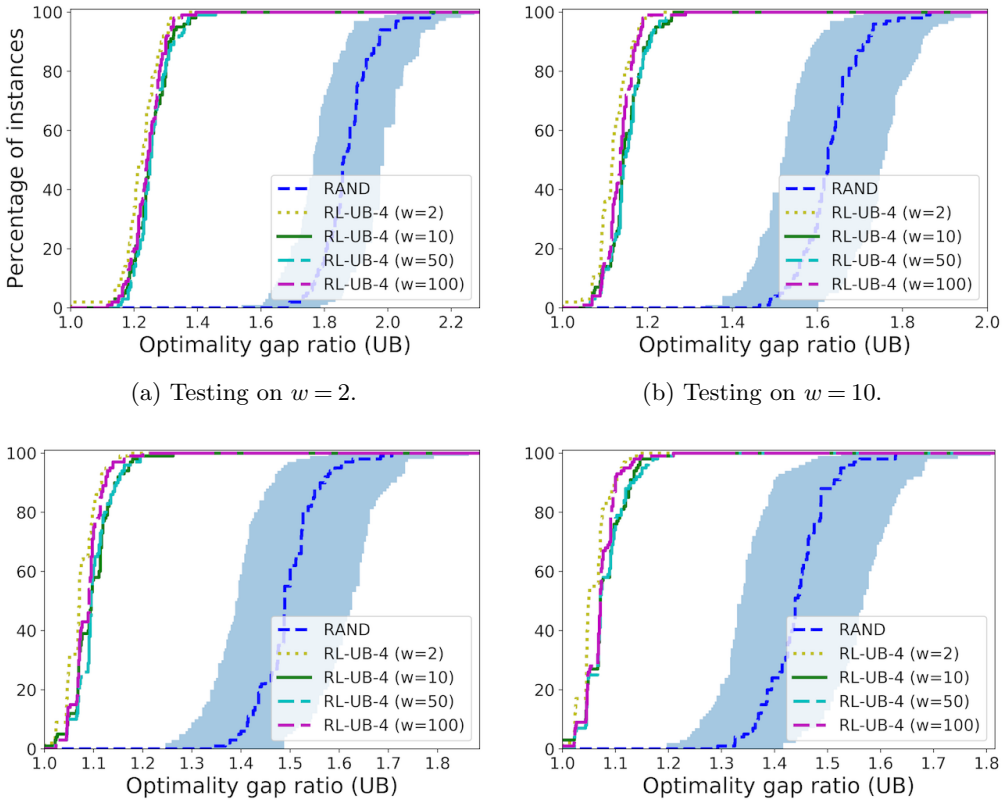
The goal of the experiments is twofold. First, we show the adequacy of the approach for computing both upper and lower bounds in different scenarios. We also consider its robustness when applied to other configurations than the one used for the training. These experiments have been reused from

our initial conference paper (Cappart et al. 2019). We then show that this learning-based bounding mechanism can be successfully integrated inside a full-fledged branch-and-bound algorithm and can be competitive with CPLEX.

5.3.1. Evaluating the DD Width for Training This experiment is designed to set an appropriate DD maximal width (w) for training the model. Let us consider $\nu = 4$ for the attachment parameter, as done by Khalil et al. (2017). We trained four models ($w = \{2, 10, 50, 100\}$) for relaxed DDs (`RL-UB-4`), and tested the models using the same values of w. Figure 6 shows the performance profiles of the models when evaluated on relaxed DDs of various widths. It illustrates the percentage of instances (y-axis) that are able to achieve a specific bound value, expressed as a ratio to the optimal solution (x-axis). In other words, the higher the curve the better. The shaded area represents the range of `RAND` performance when considering the best and worst solution obtained among the 100 trials. These results suggest that the width chosen for the training has a negligible impact on the quality of the model, even when the width considered during testing is different than that for training. As computing shallow DDs is less computationally expensive compared to those with larger widths, we selected the model trained with a width of 2 for the next experiments.

5.3.2. Comparison with Other Methods for Relaxed DDs The approach is now compared to the other variable ordering heuristics for relaxed DDs. Barabasi-Albert graphs of varying densities ($\nu = \{2, 4, 8, 16\}$) are considered, and a specific model is trained for each density. The different models are referred to as `RL-UB-`ν, where `UB` stands for the upper bound and ν the density coefficient. Results are presented in Figure 7 for relaxed DDs with a width of 100. In all tested configurations, our approach provides a better upper bound than the `RAND`, `MIN`, `MPD`, and `DEG` heuristics. For the sparsest graphs (Figure 7a), the optimal solution is reached for almost all of the instances. When the graphs are relatively sparse (Figure 7b), the linear relaxation provides the best bound. However, this trend decreases as the density of the graphs grows (Figures 7c and 7d). For these graphs, our model gives the best performance for all of the instances.

5.3.3. Evolution of the Width We focus on the situation shown in Figure 7b, where `RL-UB-4` provides a worse bound than the linear relaxation of the problem. Figure 8a depicts the evolution of the optimality gap when the model is tested on relaxed DDs of an increasingly larger width. As `RAND` provided results far outside the range of the other methods for relaxed DDs, we do not include it. We observe that `RL-UB-4` remains better than the other ordering heuristics tested, and when the DD width is sufficiently large ($w > 1000$), the linear relaxation bound is surpassed and the optimal solution is almost reached ($w = 10000$). We also notice that the second-best heuristic (`MIN`) is still worse than the linear relaxation even at this large width. Figure 8b reports the execution time of the different methods. The linear relaxation is the fastest method and is almost



(a) Testing on $w = 2$.

(b) Testing on $w = 10$.

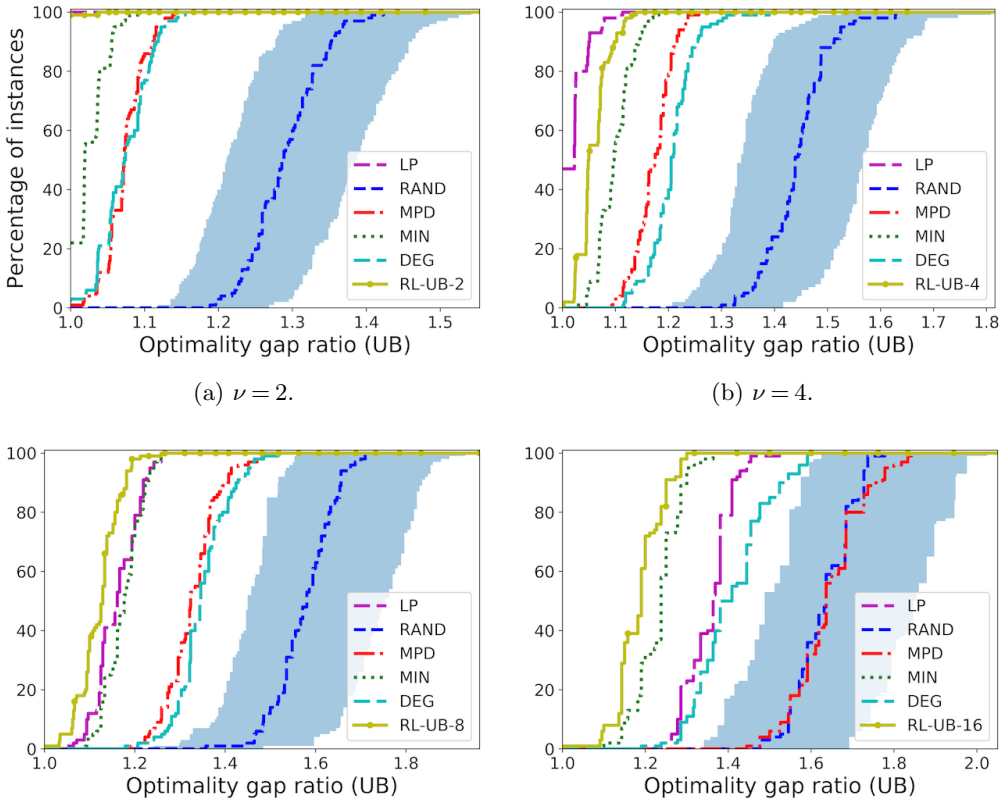
(c) Testing on $w = 50$.

(d) Testing on $w = 100$.

Figure 6 **Performance profiles of models trained with different widths for relaxed DDs. The y-axis represents the percentage of instances that are able to achieve a specific dual bound value, expressed as a ratio with the optimal solution (x-axis). Each curve (except the random one) represents the performance of a model trained with a specific maximum DD width $w \in \{2, 10, 50, 100\}$. The name `RL-UB-4` designates a model trained on graphs with attachment parameter $\nu = 4$ for generating an upper bound (`UB`). Results suggest that the DD width used for training has a negligible impact on the model performance.**

instantaneous. Concerning the heuristics, `MPD` and `DEG` are static and the execution time for each generally increases proportionally with the width, while `MIN` processes the nodes dynamically in each layer to determine the next vertex to insert, which is more expensive. In the worst case, selecting a vertex is done with a time complexity of $\mathcal{O}(w \times n)$ per layer, with w the maximum width and n the number of nodes (Bergman et al. 2012).

5.3.4. Comparison with Other Methods for Restricted DDs Experiments similar to those reported on in Figure 7 were also carried out for restricted DDs. They are referred as `RL-LB-`ν, where `LB` stands for the lower bound. Results for restricted DDs with a width of 2 are depicted in Figures 9a-9d. Again, our model has the best results from those tested and provides stronger lower bounds, close to the optimal solution. Optimality is reached for $\approx 90\%$ of the easiest instances ($\nu = 2$) and for $\approx 30\%$ of the hardest ones ($\nu = 16$). We also notice that for restricted DDs, `DEG`



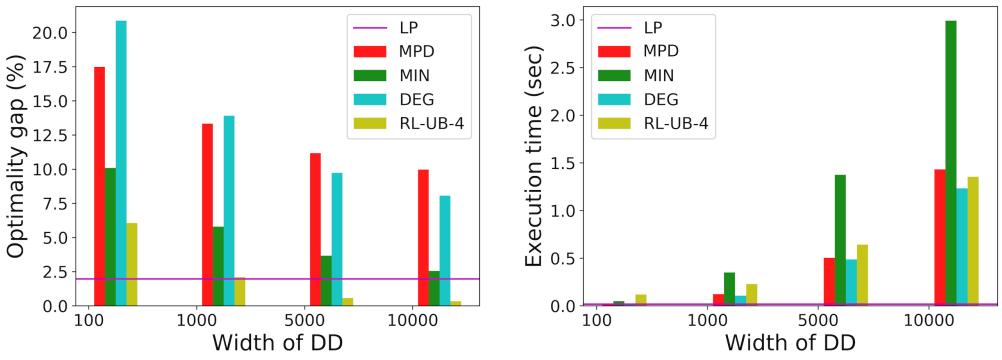
(a) $\nu = 2$. (b) $\nu = 4$.

(c) $\nu = 8$. (d) $\nu = 16$.

Figure 7 **Performance profiles on graphs of different distributions (ν) for relaxed DDs ($w = 100$). The y-axis represents the percentage of instances that are able to achieve a specific dual bound value, expressed as a ratio to the optimal solution (x-axis). Each curve represents the performance of a model. The name `RL-UB-`$\{2, 4, 8, 16\}$ designates a model trained by reinforcement learning on graphs with attachment parameter $\nu = \{2, 4, 8, 16\}$ for generating an upper bound (`UB`). A larger number indicates that the graphs are denser. These results suggest that the learned model provides the best bounds for dense graphs, while being close to the linear relaxation for sparse graphs.**

is the strongest competitor, while `MIN` is more efficient for relaxed DDs. This result indicates that the choice of a universal dedicated heuristic is not trivial.

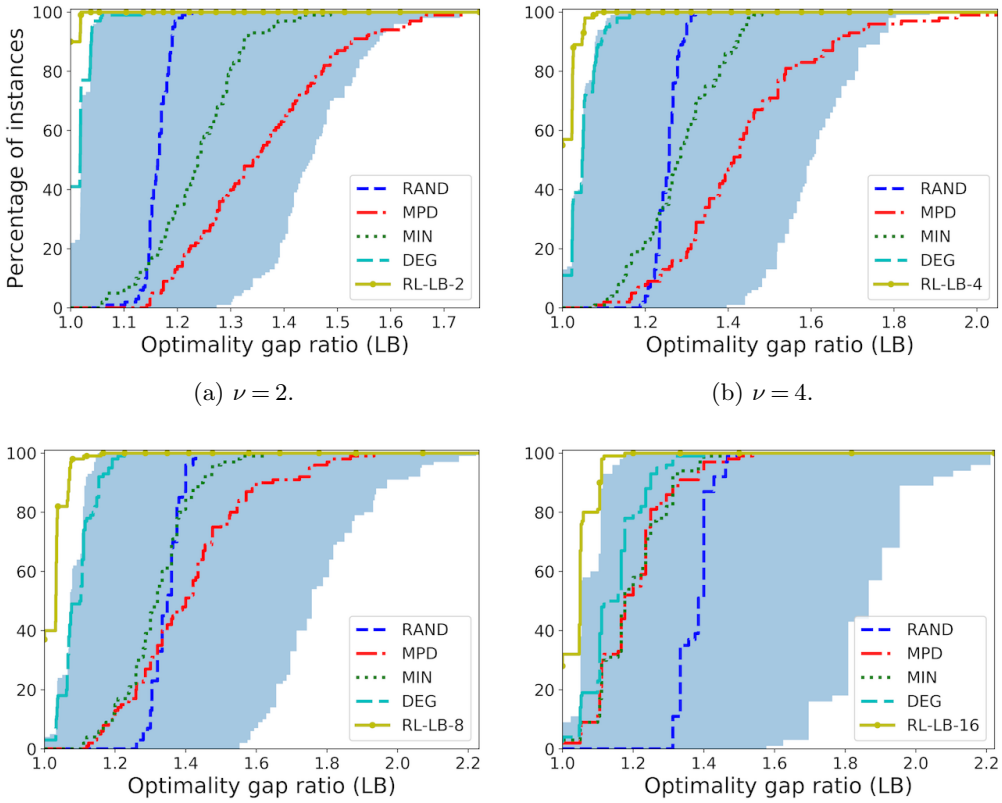
5.3.5. Analysis of the Ordering for Relaxed/Restricted DDs As shown in Figure 3, a different model is learned to obtain the variable ordering of relaxed and restricted DDs. One may wonder if a model trained using relaxed DDs can also be used to generate the ordering of restricted DDs (or the inverse), the potential benefit being that only one model would be required to compute the bounds. Figure 10a replays the experiments shown in Figure 7b but uses the model trained with restricted DDs (`RL-LB-4`). Similarly, Figure 10b replays the experiments shown in Figure 9b using `RL-UB-4`. As we can see, the models do not perform well and their performance is similar to, or even worse than, the random ordering. This result indicates that a model that is suited for relaxed DDs is not necessarily good for the restricted counterpart, and vice versa.



(a) Evolution of width. (b) Evolution of time.

Figure 8 **Quality of the upper bound in terms of optimality gap (i.e., the relative gap between the bound and the cost of the optimal solution) and the execution time required to build the DD, for different widths and ordering schemes. The inflexible bound obtained with the linear relaxation is also illustrated. Results (with $\nu = 4$) show that when the width increases, the learned model (`RL-UB-4`) can obtain significantly better bounds (left figure), but at the expense of an increased execution time (right figure).**

5.3.6. Application to a Branch-and-Bound Algorithm Following Parjadis et al. (2021), we integrate the learned bounds in a DD-based branch-and-bound algorithm (Bergman et al. 2016b) in order to assess the practical use of the framework. Compared to the work in (Parjadis et al. 2021), we conduct experiments on graphs with up to 300 nodes (instead of 120), and the caching mechanism has been dropped in favor of other improvements in the efficient RL agent integration. This algorithm works by building relaxed and restricted decision diagrams at each node of the branch-and-bound search in order to obtain both primal and dual bounds, and uses relaxed DDs for branching decisions. We propose two ways to integrate our framework within this process. In the first method, the trained models are called at each node to build the decision diagrams (`DD-RL`). Alternatively, we provide a more efficient integration designed to obtain a trade-off between the quality of the bounds and the execution time (`DD-RL+`). The rational behind this second integration is that calling a neural network many times at each node comes with significant overhead, and this provides a mechanism by which this overhead can be mitigated. The improvements of `DD-RL+` include the following: (1) only the relaxed DD is built using the trained model, (2) the model is called only for the first 50 search nodes, then the `MIN` heuristic is used, and (3) at each prediction, the next m variables are selected instead of only the single next variable. This is done by taking m variables with the highest Q-values. These three improvements were designed empirically. We observed that the primal bound has a negligible impact on the tree-search size compared to the dual bound, and that the bounds found at the beginning of the search are the most important. The last modification enables the selection of several variables with a single call to the model. It incurs a slight loss in the bound quality in exchange for a reduction in execution time by a factor



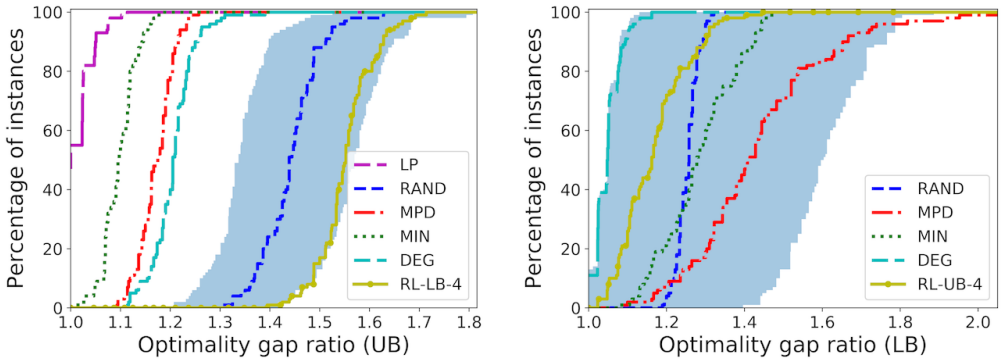
(a) $\nu = 2$. (b) $\nu = 4$.

(c) $\nu = 8$. (d) $\nu = 16$.

Figure 9 **Performance profiles on graphs of different distributions (ν) for restricted DDs ($w = 2$). The y-axis represents the percentage of instances that are able to achieve a specific primal bound value, expressed as a ratio to the optimal solution (x-axis). Each curve represents the performance of a model. The name `RL-LB-`$\{2, 4, 8, 16\}$ designates a model trained by reinforcement learning on graphs with attachment parameter $\nu = \{2, 4, 8, 16\}$ for generating a lower bound (`LB`). A larger number indicates that the graphs are denser. These results suggest that the learned model provides the best bounds in every situation studied.**

m. We set m to 5 in the reported results. Both methods are first compared to a DD-based branch-and-bound algorithm resorting to the `MIN` heuristic for the variable ordering. Results are presented in Figure 11a for the number of nodes explored before proving optimality, and in Figure 11b for the related execution time. Interestingly, we observe that even if a major reduction in the number of nodes can be obtained by `DD-RL` versus `MIN`, the execution time is dramatically worse with the learned bounds. This is due to the expensive call of the neural networks required to compute the bounds. The improvements of `DD-RL+` show both a reduction in search nodes and execution time.

5.3.7. Comparisons against CPLEX This final experiment aims to validate our approach against an industrial and efficient MIP solver (`CPLEX`), based on the linear relaxation and improved by other mechanisms such as cutting planes and pre-computation of cliques (Grötschel et al.



(a) Replay of Figure 7b. (b) Replay of Figure 9b.

Figure 10 **Performance of the model trained with restricted DDs on relaxed DDs and vice versa, reproducing the experiments of a previous situation but with another learned model. The results suggest that a model trained to obtain a tight upper bound is not efficient to obtain a tight lower bound, and conversely.**

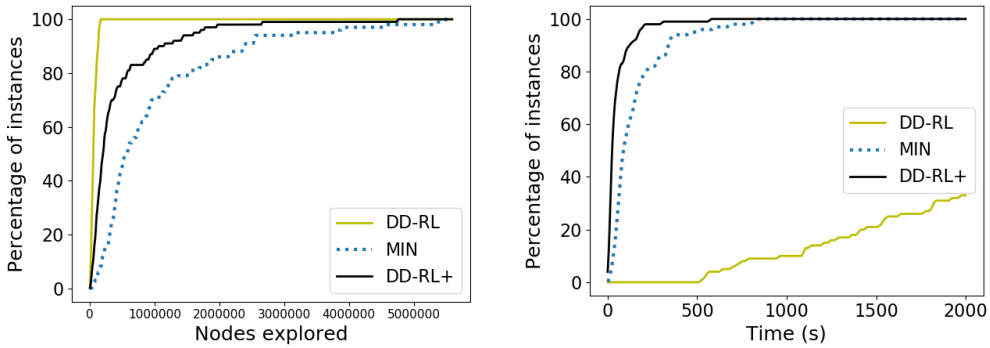


Figure 11 **Branch-and-bound results for 100 MISP instances (Barabasi-Albert graphs of 140 nodes, with $\nu = 4$). The y-axis represents the percentage of instances that are able to solve the problem at optimality, within a given number of nodes explored (x-axis). These results show that the our learned solver without improvement (`DD-RL`) is able to reduce the number of nodes explored compared to the commonly used heuristic (`MIN`), but at the expense of a significantly longer execution time. The improved version of the solver (`DD-RL+`) is able to compensate for this limitation.**

1993). The tested instances were generated randomly following a Erdos-Renyi model for graphs of $n \in \{200, 250, 300\}$ nodes and density $d = 0.3$. We made this choice as the Barabasi-Albert graphs considered so far are sparse and thus are solved efficiently with a branch-and-bound algorithm. We also increased the size of the graphs. Results are reported in Table 1 and show the number of nodes explored, the optimality gap, and the execution time. A timeout (t.o.) of 1800 seconds is used. `DD-RL` uses the learned bounds until the DDs reach a depth of 50 layers, and `DD-RL+` uses RL agent orderings for the first 100 search nodes. The model was trained on graphs of size 250 nodes. We consistently observe the superiority of `DD-RL+` in terms of execution time and optimality gap.

Table 1 Average number of nodes, optimality gap, and execution time for solving 100 MISP instances of size n. Results show the superiority of `DD-RL+` compared to CPLEX and other DD methods for the three configurations.

Algorithms	$n = 200$			$n = 250$			$n = 300$		
	Nodes (×1000)	Opt. Gap. (%)	Time (sec)	Nodes (×1000)	Opt. Gap (%)	Time (sec)	Nodes (×1000)	Opt. Gap (%)	Time (sec)
`CPLEX`	8771	0	432	27059	15	t.o.	-	40	t.o.
`DD-MIN`	998	0	66	4947	0	416	-	7.2	t.o.
`DD-RL`	800	0	180	3263	0	1140	-	65	t.o.
`DD-RL+`	879	0	61	4018	0	362	-	4.7	t.o.

5.4. Discussion

These experiments illustrate the promise of reinforcement learning for obtaining appropriate variable orderings and improving both primal and dual bounds. Interestingly, training a model using shallow DDs (width of 2) is sufficient to make these improvements. The model is also able to generalize to DDs with a larger width, producing tighter bounds. We also observed that a model trained for getting a dual bound is not suited for getting a primal one, and vice versa. We then illustrated how the bounds can be successfully integrated within a branch-and-bound algorithm and be competitive with the CPLEX branch-and-bound algorithms for dense graphs. That being said, we observed that integrating learned bound (although very tight) inside a solver is not trivial, as the time needed to call a neural network is prohibitive compared to a simpler heuristic. To tackle this issue, we proposed different mechanisms to integrate the learned bounds, resulting in a moderate loss in the bound accuracy but leading to a great improvement in time efficiency.

In order to validate the results, we carried out experiments on three other problems (*maximum cut*, *set covering*, and *0-1 knapsack problem*). The experimental protocol and detailed results are shown in the online supplement. The main insights we gained from these experiments is that learning is effective and provides bounds that are at least as tight as the ones obtained with a standard ordering heuristic for DDs. Specifically, state-of-the-art orderings are achieved for the *maximum cut* problem. We note that the training phase is more expensive, and that a larger DD width must be considered for this problem.

The *set covering* case study can also benefit from the integrated framework. Experiments on relaxed and restricted DDs both show that the learning is effective and improves the bounds compared to classic heuristics in most situations. Although the relaxation bounds obtained are close to the ones obtained by the linear relaxation, we gain flexibility. Since the relaxation bounds provided by DDs are flexible, they can be easily improved by increasing the width of the DD. Such an improvement would not be possible with the linear relaxation bounds.

Finally, *0-1 knapsack* experiments demonstrate that even if learning is effective and can be done without decreasing performance, it may not be useful for this problem. Our first observation

indicates that the recursive formulation for this problem allows greedy choices (i.e., selecting the item with the best utility/weight ratio) to obtain good performances, perhaps close to the best ordering that can be reached given the DD width chosen. Although the reinforcement learning framework is able to achieve similar performances as these heuristics, it seeks to improve upon them. A second observation is that the linear relaxation of the standard problem is superior to its DD counterpart. This is not a limitation of the reinforcement learning framework, but of DD technology itself: it appears that the bounds that can be obtained with DD do not compete with those obtained by the linear relaxation for this problem.

6. Conclusion

Prescriptive analytics provide organizations with scalable software for large-scale, automated decision-making. A large portion of prescriptive analytic methodology is based on combinatorial optimization. In order to reach as many practitioners as possible, generic optimization solvers have developed scalable implementations of state-of-the-art algorithms. One challenge in the design of scalable algorithms is the efficient computation of tight optimization bounds. Recently introduced, decision diagrams (DDs) provide a novel and flexible mechanism for obtaining high-quality and flexible bounds. The main contribution of this work is a generic framework based on deep reinforcement learning for improving the optimization bounds of dynamic programs, thanks to DD technology. This is done by learning appropriate variable orderings that are shown experimentally to tighten the bounds proven by restricted and relaxed DDs. Compared to most of the related work using machine learning for computing primal solutions, our framework (1) is generic in the sense that it can be applied to any problem that can be encoded by a recursive formulation and (2) provides flexible bounds. Similarly, it can also be used for computing dual bounds, provided that a relaxation operator has been defined.

The general idea of the framework is to exploit a recursive formulation of an optimization problem, which can be used to define the reinforcement learning environment and the DD construction at the same time. In doing so, it links the areas of operations research and machine learning, using the recursive formulation as a bridge between both fields. Linking both fields in a generic way is a hot topic in both research communities and remains a challenge. This contribution embraces that challenge and moves the discussion forward. Experimental results indicate the promise of our approach when integrated within a branch-and-bound algorithm.

References

Albert R, Barabási AL (2002) Statistical mechanics of complex networks. *Reviews of modern physics* 74(1):47.

Andersen HR, Hadzic T, Hooker JN, Tiedemann P (2007) A constraint store based on multivalued decision diagrams. *International Conference on Principles and Practice of Constraint Programming*, 118–132 (Springer).

Bello I, Pham H, Le QV, Norouzi M, Bengio S (2016) Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* .

Bengio Y, Lodi A, Prouvost A (2018) Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128* .

Bergman D, Cire AA (2018) Discrete nonlinear optimization by state-space decompositions. *Management Science* 64(10):4700–4720.

Bergman D, Cire AA, van Hoeve WJ, Hooker J (2016a) *Decision diagrams for optimization* (Springer).

Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2012) Variable ordering for the application of BDDs to the maximum independent set problem. *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, 34–49 (Springer).

Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2013) Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing* 26(2):253–268.

Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2016b) Discrete optimization with decision diagrams. *INFORMS J. on Computing* 28(1):47–66, ISSN 1526-5528.

Bergman D, van Hoeve WJ, Hooker JN (2011) Manipulating MDD relaxations for combinatorial optimization. Achterberg T, Beck JC, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 20–35.

Berthold T (2006) Primal heuristics for mixed integer programs.

Bryant RE (1986) Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100(8):677–691.

Cappart Q, Chételat D, Khalil E, Lodi A, Morris C, Veličković P (2021a) Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544* .

Cappart Q, Goutierre E, Bergman D, Rousseau LM (2019) Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 33(01):1443–1451.

Cappart Q, Moisan T, Rousseau LM, Prémont-Schwarz I, Cire AA (2021b) Combining reinforcement learning and constraint programming for combinatorial optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 3677–3687.

Christofides N, Mingozzi A, Toth P (1981) State-space relaxation procedures for the computation of bounds to routing problems. *Networks* 11(2):145–164.

Dai H, Dai B, Song L (2016) Discriminative embeddings of latent variable models for structured data. *International Conference on Machine Learning*, 2702–2711.

Deudon M, Cournut P, Lacoste A, Adulyasak Y, Rousseau LM (2018) Learning heuristics for the TSP by policy gradient. *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 170–181 (Springer).

Dolan ED, Moré JJ (2002) Benchmarking optimization software with performance profiles. *Mathematical programming* 91(2):201–213.

Farràs O, Kaced T, Martín S, Padró C (2018) Improving the linear programming technique in the search for lower bounds in secret sharing. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 597–621 (Springer).

Gamrath G, Koch T, Martin A, Miltenberger M, Weninger D (2015) Progress in presolving for mixed integer programming. *Mathematical Programming Computation* 7(4):367–398.

Gasse M, Chételat D, Ferroni N, Charlin L, Lodi A (2019) Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems* 32.

Grötschel M, Lovasz L, Schrijver A (1993) *Geometric Algorithms and Combinatorial Optimization* (Springer).

Hadzic T, Hooker J (2006) Postoptimality analysis for integer programming using binary decision diagrams. *GICOLAG Workshop*.

Hagberg A, Swart P, S Chult D (2008) Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

He H, Daume III H, Eisner JM (2014) Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 3293–3301.

Henderson P, Islam R, Bachman P, Pineau J, Precup D, Meger D (2018) Deep reinforcement learning that matters. *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. *International conference on learning and intelligent optimization*, 507–523 (Springer).

Khalil E, Dai H, Zhang Y, Dilkina B, Song L (2017) Learning combinatorial optimization algorithms over graphs. *Advances in Neural Information Processing Systems*, 6351–6361.

Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

Kool W, van Hoof H, Welling M (2019) Attention, learn to solve routing problems! *International Conference on Learning Representations*.

LeCun Y, Bengio Y, Hinton G (2015) Deep learning. *nature* 521(7553):436.

Lee CY (1959) Representation of switching circuits by binary-decision programs. *Bell system Technical journal* 38(4):985–999.

Lin LJ (1992) Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* 8(3-4):293–321.

Lodi A, Zarpellon G (2017) On learning and branching: a survey. *TOP* 25(2):207–236.

Masters D, Luschi C (2018) Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612* .

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G, et al. (2015) Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Parjadis A, Cappart Q, Rousseau LM, Bergman D (2021) Improving branch-and-bound using decision diagrams and reinforcement learning. *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 446–455 (Springer).

Riedmiller M (2005) Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. *European Conference on Machine Learning*, 317–328 (Springer).

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. *nature* 323(6088):533.

Sadykov R, Vanderbeck F, Pessoa A, Tahiri I, Uchoa E (2019) Primal heuristics for branch and price: The assets of diving methods. *INFORMS Journal on Computing* 31(2):251–267.

Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2008) The graph neural network model. *IEEE transactions on neural networks* 20(1):61–80.

Schaul T, Quan J, Antonoglou I, Silver D (2015) Prioritized experience replay. *arXiv preprint arXiv:1511.05952* .

Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van Den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, et al. (2016) Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484–489.

Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, Lanctot M, Sifre L, Kumaran D, Graepel T, et al. (2017) Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* .

Sutton RS, Barto AG (2018) *Reinforcement learning: An introduction* (MIT press).

Tang Y, Agrawal S, Faenza Y (2020) Reinforcement learning for integer programming: Learning to cut. *International Conference on Machine Learning*, 9367–9376 (PMLR).

Van Hasselt H, Guez A, Silver D (2016) Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Verhaeghe H, Lecoutre C, Schaus P (2018) Compact-mdd: efficiently filtering (s) mdd constraints with reversible sparse bit-sets. *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 1383–1389 (AAAI Press).

Watkins CJ, Dayan P (1992) Q-learning. *Machine learning* 8(3-4):279–292.