Improving Optimization Bounds Using Machine Learning: Decision Diagrams Meet Deep Reinforcement Learning

Quentin Cappart[1,4], Emmanuel Goutierre[2], David Bergman[3] and **Louis-Martin Rousseau[1]**
[1]Ecole Polytechnique de Montréal, Montreal, Canada
[2]Ecole Polytechnique, Paris, France
[3]University of Connecticut, Stamford, CT 06901, USA
[4]Element AI, Montreal, Canada
{quentin.cappart, louis-martin.rousseau}@polymtl.ca
emmanuel.goutierre@polytechnique.edu
david.bergman@uconn.edu

Abstract

Finding tight bounds on the optimal solution is a critical element of practical solution methods for discrete optimization problems. In the last decade, decision diagrams (DDs) have brought a new perspective on obtaining upper and lower bounds that can be significantly better than classical bounding mechanisms, such as linear relaxations. It is well known that the quality of the bounds achieved through this flexible bounding method is highly reliant on the ordering of variables chosen for building the diagram, and finding an ordering that optimizes standard metrics is an NP-hard problem. In this paper, we propose an innovative and generic approach based on deep reinforcement learning for obtaining an ordering for tightening the bounds obtained with relaxed and restricted DDs. We apply the approach to both the Maximum Independent Set Problem and the Maximum Cut Problem. Experimental results on synthetic instances show that the deep reinforcement learning approach, by achieving tighter objective function bounds, generally outperforms ordering methods commonly used in the literature when the distribution of instances is known. To the best knowledge of the authors, this is the first paper to apply machine learning to directly improve relaxation bounds obtained by general-purpose bounding mechanisms for combinatorial optimization problems.

Introduction

Relaxation bounds, and mechanisms by which those bounds can be improved, are perhaps the most critical element of scalable generic algorithms for discrete optimization problems. As machine learning popularizes, a natural question arises: how can machine learning be used for improving optimization bounds ? Finding a way to utilize the power of machine learning to prove tighter relaxation bounds may be a key for unlocking significant performance improvements in optimization solvers. This paper provides, to the best knowledge of the authors, a first effective approach in the literature towards achieving this goal.

The challenge one faces in using machine learning to tighten relaxation bounds is that the bound provided by classical methods (e.g., LP or SDP relaxations) are *inflexible*; the algorithm used to solve the relaxation has no effect on the quality of the bound. For example, given an IP model, the LP relaxation will report the same bound independent of what method is used to solve the relaxation and any other decision employed during the solution algorithm.

Contrastingly, approximate decision diagrams (DDs) (Bergman, van Hoeve, and Hooker 2011), a recently introduced optimization technology, provide a *flexible* bounding method, in that decisions employed in the execution of the algorithms used to build the DDs directly affect the quality of the bound. This is true for both relaxed DDs, that prove relaxation bounds, and restricted DDs, that identify primal solutions. This opens the door for potential integration with machine learning.

Initially introduced for representing switching circuits (Lee 1959) and for formal verification (Bryant 1986), DDs in discrete optimization are used to encode the feasible solutions of a problem while preserving its combinatorial structure. A common application is to provide bounds, both upper and lower, for discrete optimization problems (Bergman, van Hoeve, and Hooker 2011; Bergman et al. 2013). However, the quality of the bounds is known to be tightly related to the variable ordering considered during the construction of the DD (Bergman et al. 2012). It has been shown that finding an optimal ordering for general DDs is NP-hard and is often challenging to even model. Besides, improving a given variable ordering is known to be NP-complete (Bollig and Wegener 1996). Thus, designing methods for finding a good ordering is a hot topic in the community and continues as a challenge. The idea suggested in this paper is to use machine learning to identify good variable orderings that therefore result in tighter objective function bounds.

In another field of research, reinforcement learning (RL) (Sutton and Barto 1998) is an area of machine learning focusing on how an agent can learn from its interactions with an environment. The agent moves from state to state by performing a sequence of actions, each of them giving a specific reward. The behavior of an agent is characterized by a policy, determining which action should be taken from each state. Given this context, the goal is to learn a policy maximizing the sum of rewards of each action done by the agent.

However, traditional methods for RL 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 considered. Recently, deep learning (LeCun, Bengio, and Hinton 2015) provided new tools to overcome this problem. The idea is to use a deep architecture as a function approximation for generalizing knowledge from visited to unknown states. Such an improvement enabled RL to scale to problems that were previously intractable. Notorious examples are the superhuman performances obtained for the game of Go (Silver et al. 2016) and Atari 2600 (Mnih et al. 2013; Mnih et al. 2015). The combination of RL with a deep network is commonly referred as *deep reinforcement learning* (DRL) (Arulkumaran et al. 2017).

Even more recently, DRL has also been applied to identify high-quality primal bounds to some NP-hard combinatorial problems. Most work focuses on the classical Traveling Salesman Problem (Bello et al. 2016; Deudon et al. 2018), with the exception of the approach of Khalil et al. (Khalil et al. 2017) that tackles four NP-hard problems having a graph structure. They use a deep learning architecture in order to embed the graph structure into features (Dai, Dai, and Song 2016). The competitive results obtained suggest that this approach is a promising new tool for finding solutions to NP-hard problems. In this paper, we further push these efforts to be able to generate dual bounds.

Given this related work, our contribution is positioned as follows. We propose a generic approach based on DRL in order to identify variable orderings for approximate DDs. The goal is to find orderings providing tight bounds. The focus is on relaxed DDs, as this provides a mechanism for utilizing machine learning to improve relaxation bounds, but we also show the effectiveness for restricted DDs, adding to the recent literature on using machine learning for finding high-quality heuristic solutions. The approach has been validated on the Maximum Independent Set Problem, for which the variable ordering has been intensively studied (Bergman et al. 2012). Its application to the Maximum Cut Problem is also considered. We note that there has been limited work on applying machine learning to identify variable orderings for DDs in unrelated fields (Carbin 2006; Grumberg, Livne, and Markovitch 2011). To the best of our knowledge, this work has not been extended to optimization.

This paper is structured as follows. The next section introduces the technical background related to DDs and RL. The process that we designed for learning an appropriate variable ordering is then presented. The RL model and the learning algorithms are detailed and the construction of the DD using RL is described. Finally, experiments on synthetic instances are carried out in the last section.

Technical Background

Decision Diagrams

In the optimization field, a *decision diagram* (DD) is a structure that encodes a set of solutions to a constrained optimization problem (COP) $\langle X, D, C, O\rangle$ where X is the set of variables, D the set of discrete domains restricting the values that variables $x \in X$ can take, C the sets of constraints and O the objective function. Formally, a DD associated to a combinatorial problem P is a directed-layered-acyclic graph $B_P = \langle U, A, d\rangle$ where U is the set of nodes, A the set of arcs and d is a function $A \rightarrow \mathbb{N}$ associating a label at each arc. The set of nodes U is partitioned into layers L_i, i.e., $U = \cup_{i=1}^{m} L_i$. Layers L_1 and L_m are both composed of a single node: the root and the terminal node, respectively. The width $w_i(B)$ of layer L_i in a DD B is defined as the number of nodes in that layer: $w_i(B) = |L_i|$. The width $w(B)$ of the DD is the maximum-width layer: $w(B) = \max_i w_i(B)$. Each arc $a \in A$ is directed from a node in a layer L_i to a node in layer L_{i+1} where $i \in [0, m-1]$. The function d associates to each arc a a *label* $d(a)$. The arcs directed out of each node $u \in U$ have distinct labels, i.e., at most one arc with tail a having any domain value d. We assume that for each u, there must exist a directed path from the root node to u and from u to the terminal node. A *cost* $c(a) \in \mathbb{R}$ is also associated to every arc in a, which is used to encode the objective function of solutions.

In this paper, a DD B_P for a COP P has $n+1$ layers where n is the number of decision variables in P. Each layer L_i (except the last one) is linked to one variable x_i of P and an arc a from L_i to L_{i+1} with label $d(a)$ represents the assignment x_i to $d(a)$. A direct path from the root to the terminal node of B_P corresponds then to a solution of P. The assignment of variables in P to layers during the construction of the DD is referred as the *variable ordering*.

A DD is *exact* when the solutions encoded align exactly with the feasible solutions of the initial problem P and for any arc-directed root-to-terminal node path p, the sum of the costs of the arcs equates to the evaluation of the objective function for the solution x it corresponds, i.e., $\sum_{a \in p} c(a) = O(x)$. In this case, the longest path (assuming a maximization problem) from the root to the terminal node corresponds to the optimal solution of P. However, the width of DDs tends to grow exponentially with the number of variables in the problem, which reduces its usability for large instances. A DD is *relaxed* when its encodes a superset of the feasible solutions of P and for any arc-directed root-to-terminal node path p, the sum of the costs of the arcs is an upper bound (still in the case of a maximization problem) on the evaluation of the objective function for the solution x it corresponds, i.e., $\sum_{a \in p} c(a) \geq O(x)$. A relaxed DD can be constructed incrementally, merging nodes on each layer until the width is below a threshold (Bergman, van Hoeve, and Hooker 2011) in such a way that no solution is lost during the merging process. Hence, for a relaxed DD, the longest path gives an upper bound of the optimal solution for P. Finally, a DD is *restricted* when it under-approximates the feasible solutions of P and for any arc-directed root-to-terminal node path p, the sum of the costs of the arcs is a lower bound on the evaluation of the objective function for the solution x it corresponds, i.e., $\sum_{a \in p} c(a) \leq O(x)$. There are several ways to construct such a DD, perhaps the simplest one is removing nodes from each layer once the width threshold is reached. Unlike relaxed DDs, solutions are lost during the reduction. For maximization problems, the longest path provides a lower bound of the optimal solution for P. Optimization bounds can thereby be directly obtained through relaxed

and restricted DDs. Both take as input a specified maximum width, and it has been empirically shown that larger DDs generally provide tighter bounds but are in return more expensive to compute. An exhaustive description of DDs and their construction are provided in (Bergman et al. 2016).

Reinforcement Learning

Let $\langle S, A, T, R\rangle$ be a tuple representing a deterministic couple agent-environment where S is the set of states in the environment, A the set of actions that the agent can do, $T : S \times A \to S$ the transition function leading the agent from a state to another one given the action taken and $R : S \times A \to \mathbb{R}$ the reward function of taking an action from a specific state. 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 $t \in [1, \Theta]$. Such a sequence is called an *episode* where s_Θ is the terminal state. The expected return after time step t is denoted by $G_t = \sum_{k=t+1}^{\Theta} \gamma^{k-t-1} R(s_k, a_k)$ where $\gamma \in [0, 1]$ is a discounting factor used for parametrizing the weight of future rewards. 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 problem is to find a policy maximizing the expected return: $\pi^\star = \operatorname{argmax}_\pi Q^\pi(s, a)\ \forall s \in S, \forall a \in A$. In practice, $\pi^\star$ is computed from an initial policy and by two nested operations: (1) the policy evaluation, making the action-value function consistent with the current policy, and (2) the policy iteration, improving greedily the current policy.

However, in practice, 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 then required. Instead of computing the optimal action-value function, Q-learning approximates the function by iteratively updating a current estimate after each action. The update function is defined as follows: $Q(s_t, a_t) \xleftarrow{\alpha} R(s_t, a_t) + \gamma \max_{a \in A} Q(s_{t+1}, a)$, where $x \xleftarrow{\alpha} y$ denotes the update $x \leftarrow x + \alpha(y - x)$ and $\alpha \in (0, 1]$ the learning rate.

Another issue arising 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 them, 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 (Bottou 2010) or another optimizer coupled with back-propagation (Rumelhart, Hinton, and Williams 1986) is then used for updating $\mathbf{w}$ and aims to minimize the squared loss between the current Q-value and the new value that should be assigned using Q-learning: $\mathbf{w} \leftarrow \mathbf{w} - \frac{1}{2}\alpha \nabla L(\mathbf{w})$ where the square loss is $L(\mathbf{w}) = \big(R(s_t, a_t) + \gamma \max_{a \in A} \widehat{Q}(s_{t+1}, a, \mathbf{w}) - \widehat{Q}(s, a, \mathbf{w})\big)^2$. Updates are done using *experience replay*. Let $\langle s, a, r\rangle$ be a sample representing an action done at a specific state with its reward and $\mathcal{D}$ a sample store. Each time an action is performed, $\langle s, a, r\rangle$ is added in $\mathcal{D}$. Then, the optimizer updates $\mathbf{w}$ using a random sample taken from $\mathcal{D}$.

Learning Process

Reinforcement Learning Formulation

Designing a RL model for determining the variable ordering of a DD associated to the COP $\langle X, D, C, O\rangle$ requires defining, adequately, the tuple $\langle S, A, T, R\rangle$ to represent the system. Our model is defined as follows.

State A state $s \in S$ is a pair $\langle s_L, s_B\rangle$ containing an ordered sequence of variables s_L and a partially constructed DD s_B associated with variables in s_L. A state s is terminal if s_L includes all the variables of X.

Action An action is defined as the selection of a variable from X. An action a_s can be performed at state s if and only if it is not yet inserted in s_B ($a_s \in X \setminus s_L$).

Transition A transition is a function updating a 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 the sequence y, we have $T(s, a_s) = \langle s_L :: a_s, s_B \oplus a_s\rangle$.

Reward The reward function is designed to tighten the bounds obtained with the DD. When maximizing, 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 DD 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: $R^{ub}(s, a_s) = -\big(\lceil s_B\rceil - \lceil s_B \oplus a_s\rceil\big)$. It is rewarded for the lower bound where we want to increase it instead: $R^{lb}(s, a_s) = \big(\lfloor s_B\rfloor - \lfloor s_B \oplus a_s\rfloor\big)$. For minimization problems, the shortest path must be considered instead. The upper bounds are then provided by restricted DDs and lower bounds by relaxed DDs.

Note that this formalization is generic and can be applied to any problem that can be represented by a DD constructed layer-by-layer. Indeed, all the problem-dependent characteristics are embedded into the DD construction and the insertion of variables (operator $\oplus$ in the transition function). It is interesting to see that the construction of a DD can be elegantly formulated as a RL problem. Indeed, both methods are based on dynamic programming and a recursive formulation which ease their integration.

Learning Algorithm

The basis of the learning algorithm relies on neural fitted Q-learning as described in the previous section and is presented in Algorithm 1. At each iteration, a COP (P) is randomly taken from the training set and the learning is conducted on

it. 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 (formally defined later), other instances from that class of problem should be used during the training. The algorithm returns a vector of weights ($\mathbf{w}$) which is used for parametrizing the approximate action-value function $\widehat{Q}$. The basic algorithmic framework can be improved through the following:

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. 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 a follows: $\mathbf{w} \leftarrow \mathbf{w} - \frac{1}{2N}\alpha \sum_{j=1}^{N} \nabla L_j(\mathbf{w})$.

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 increasingly favoring exploitation.

Reward scaling Gradient-based methods have difficulties to learn 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, as stressed by (Henderson et al. 2017), tiny rewards can also lead to failed learning. We let $\rho \in \mathbb{R}$ be the scaling factor, generally defined as a power of 10, and rescale the rewards as $r_t = \rho R(s_t, a_t)$.

Algorithm 1: Learning Algorithm.

▷ **Pre:** $\langle S, A, T, R\rangle$ is an environment tuple.
▷ $\mathcal{M}$ is the training set containing COPs.
▷ K is the number of iterations.
▷ N is the batch size.
▷ Θ is the length of an episode.
▷ $\epsilon, \rho, \alpha, \gamma$ are parameters as defined previously.
▷ π, $\mathbf{w}$ are randomly initialized.

$\mathcal{D} := \emptyset$ ▷ *Experience replay store*
for i **from** 1 **to** K **do**
  $P := \text{randomValueFrom}(\mathcal{M})$
  $\langle S, A, T, R\rangle := \text{initializeEnvironment}(P)$
  $s_1 := \langle \emptyset, \emptyset \rangle$ ▷ *Decision diagram is empty*
  for t **from** 1 **to** Θ **do**
    $\pi := \text{argmax}_{\pi}\, \widehat{Q}^{\pi}(s, a, \mathbf{w}) \quad \forall s \in S, \forall a \in A$
    $k := \text{randomValueFrom}([0, 1])$
    if $k > \epsilon$ **then**
      $a_t := \pi(s_t)$ ▷ *Following policy*
    else
      $a_t := \text{randomValueFrom}(A)$ ▷ *ϵ-greedy*
    $r_t := \rho R(s_t, a_t)$ ▷ *Reward scaling*
    $s_{t+1} := T(s_t, a_t)$
    $\mathcal{D} := \mathcal{D} \cup \{\langle s_t, a_t, r_t\rangle\}$ ▷ *Store update*
    for j **from** 1 **to** N **do**
      $e := \text{randomValueFrom}(\mathcal{D})$
      $L_j(\mathbf{w}) :=$ square loss using e and γ
    $\mathbf{w} := \mathbf{w} - \frac{1}{2N}\alpha \sum_{j=1}^{N} \nabla L_j(\mathbf{w})$ ▷ *Mini-batch*
    update(ϵ)
return $\mathbf{w}$

Decision Diagram Construction

Once the model has been trained, the next step is to use it in order to build the DD for a new instance. Let us illustrate the construction on the *Maximum Independent Set Problem*.

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 is no two vertices in I that are connected by an edge of E. The Maximum Independent Set Problem (MISP) consists in finding the independent set with the largest cardinality.*

Note that we use the term *vertices* for elements of the graph and *nodes* for DDs. The problem is fully represented by a graph $G(V, E)$ and a classical formulation assigns a binary variable x_i for each vertex $i \in V$ indicating if the variable is selected in the set or not. More details about the internal operations of the construction are provided in (Bergman et al. 2012). Specific to the learning, the environment tuple is generated for G, and the learned model is then applied on it. The environment is directly infered from the previous formulation: the current state is the list of variables x_i already considered with the DD currently built, an action consists in choosing a new variable and the transition function with the reward is associated to the DD construction. At each state, the model is called in order to compute the estimated $\widehat{Q}$-value for each action that can be performed in the current state. The network *structureToVec* (Dai, Dai, and Song 2016) can be used for parametrizing $\widehat{Q}$ (Khalil et al. 2017). The construction is driven by the policy $\pi = \text{argmax}_a\, \widehat{Q}(s, a)$. The best vertex according to the approximated action-value function is inserted in the DD at each step.

This process is illustrated in Figure 1 for a MISP instance. Solid arcs indicate that the vertex related to the current variable is selected in the solution while dashed arcs indicate the opposite. The partially constructed DD and the inserted/remaining vertices are depicted for each state. The value in each vertex indicates the $\widehat{Q}$-value computed by the model for each state-action pair. Gray vertices are the ones that are greedily selected by the policy. No vertices can be inserted twice. The construction is terminated when all vertices are inserted.

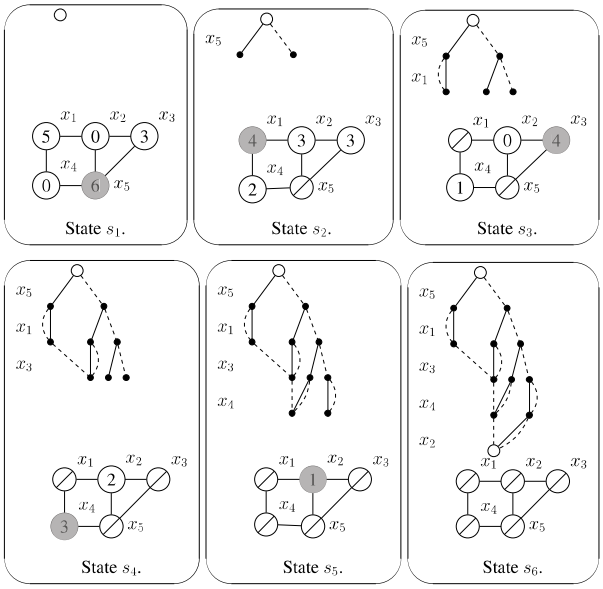


Figure 1: Example of an exact DD construction for a MISP instance, following policy $\pi = \text{argmax}_a Q^\pi(s, a)$.

Experimental Results

Our first set of experiments are carried out on the MISP, for which the impact of variable ordering has been deeply studied (Bergman et al. 2013). The last experiments analyze the generalization of the approach on the Maximum Cut Problem. For the MISP, the approach is compared with the linear programming relaxation bound, random orderings, and three ordering heuristics commonly used in the literature:

1. *Linear Programming Relaxation* (`LP`): The value of the linear relaxation obtained using a standard clique formulation for the MISP as described in (Bergman et al. 2013).
2. *Random Selection* (`RAND`): An ordering of the vertices is drawn uniformly at random from all permutations. For each test, 100 random trials are performed and the average, best and worst results are reported.
3. *Maximal Path Decomposition* (`MPD`): A *maximal path decomposition* is precomputed 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.
4. *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 minimize greedily the size of the subsequent layer.
5. *Minimum Vertex Degree* (`DEG`): The vertices are ordered in ascending order of vertex degree. The vertices with the lowest degree are inserted first.

Experimental Protocol

MISP instances were generated using the Barabasi-Albert (BA) model (Albert and Barabási 2002). Such a model is commonly used for generating real-world and scale-free graphs. They are defined by the number of nodes (n) and an attachment parameter (ν). The greater is ν, the denser is the graph. Edges are added preferentially to nodes having a higher degree. Training has been carried out on Compute Canada Cluster[1]. Training time is limited to 20 hours, memory consumption to 64 GB and one GPU (NVIDIA P100 Pascal, 12GB HBM2 memory) is used. For each configuration, the training is done using 1000 generated random BA graphs (between 90 and 100 nodes) that are refreshed every 5000 iterations. Different models with a specific value for the attachment parameter ($\nu = \{2, 4, 8, 16\}$) are trained. The model selected is the one giving the best average reward on a validation set composed of 100 graphs having the same configuration as the training graphs. The training time required to get this model is dependent on the configuration considered. It varies between 20 minutes for the best case and 6 hours for the worst case. At the first time, testing is carried out on 100 other random graphs of the same size and having the same attachment parameter as for the training. Other configurations are then considered. Performance profiles (Dolan and Moré 2002) are used for comparing the approaches. 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).

Our model is implemented upon the code of Dai et al.[2] for the learning part and upon the code of Bergman et al. (Bergman et al. 2013) for building the DDs of the MISP instances. Evaluation of the different orderings is also done using this software. The learning is done using Adam optimizer (Kingma and Ba 2014). Library `networkX` (Hagberg, Swart, and S Chult 2008) is used for generating the random graphs. For the reproducibility of results, the implementation of our approach is available online[3]. Optimal solutions of the MISP instances and the linear relaxations have been obtained using CPLEX 12.6.3.

Results

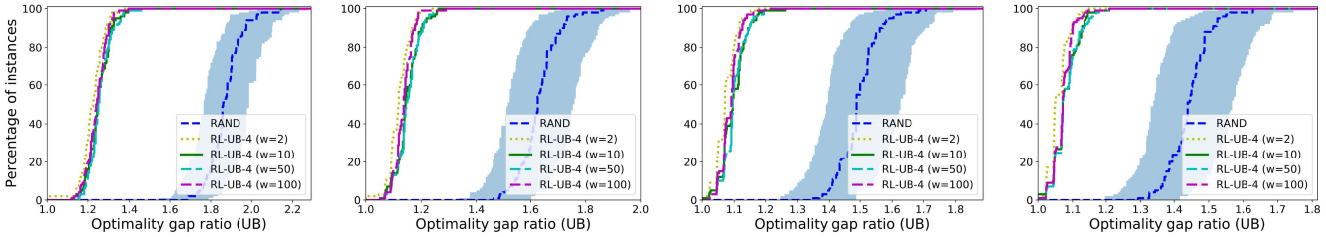
The goal of the experiments is to show the adequacy of our approach for computing both upper and lower bounds in different scenarios commonly considered in practice.

Evaluating the DD Width for Training The first set of experiments aim to determine the best DD maximal width (w) for training the model. Let us first consider $\nu = 4$ for the attachment parameter as in (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 2 shows the performance profiles of the models when evaluated on relaxed DDs of a various width. Random ordering (`RAND`) is also reported and is outperformed by the four models. The shaded area represent the range of the `RAND` performance when considering the best and the worst solution obtained among the 100 trials. Interestingly, these results suggest that the width chosen for the training has a

[1] https://www.computecanada.ca/research-portal/

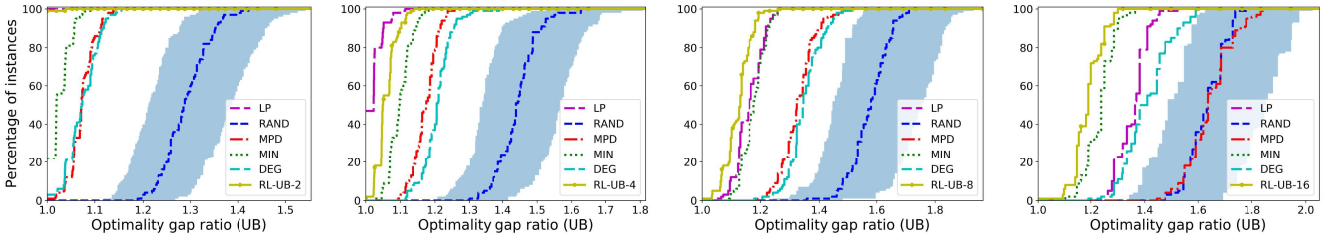
[2] https://github.com/Hanjun-Dai (graph_comb_opt)

[3] https://github.com/qcappart/learning-DD



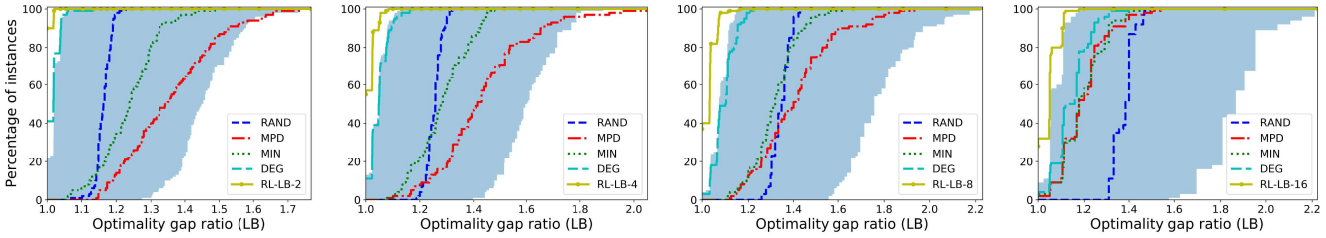
(a) Testing on $w = 2$. (b) Testing on $w = 10$. (c) Testing on $w = 50$. (d) Testing on $w = 100$.

Figure 2: Performance profiles of model trained with different widths for relaxed DDs.



(a) $\nu = 2$. (b) $\nu = 4$. (c) $\nu = 8$. (d) $\nu = 16$.

Figure 3: Performance profiles on graphs of different distributions (ν) for relaxed DDs ($w = 100$).



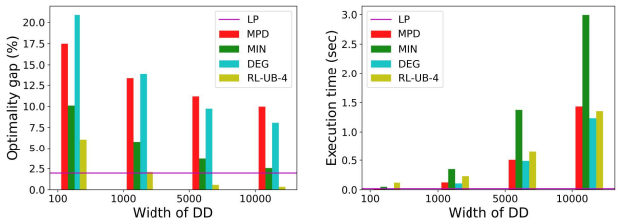
(a) $\nu = 2$. (b) $\nu = 4$. (c) $\nu = 8$. (d) $\nu = 16$.

Figure 4: Performance profiles on graphs of different distributions (ν) for restricted DDs ($w = 2$).

negligible impact on the quality of the model, even when the width considered during the testing is different than that for the training. As computing small-width DDs is less computationally expensive than those with larger widths, we select the model trained with a width of 2 for the remainder of the experiments on MISP. Concerning restricted DDs, as shown in the next set of experiments (Figures 4a-4d), lower bounds close to the optimal solutions are already obtained with small-width DDs ($w = 2$). This independence of the width chosen during the training is the most surprising result that we get through our set of experiments. Perhaps one reason of this stability is due to the merging heuristic that remains the same in all the configuration, but an in-depth explanation of these results is still an open question.

Comparison with Other Methods Our approach is now compared to the other variable ordering heuristics using BA graphs having a varied density ($\nu = \{2, 4, 8, 16\}$). A specific model is trained for each distribution for both relaxed (`RL-UB-`ν) and restricted DDs (`RL-LB-`ν). Evaluation is done on graphs following the same distribution as those used in training. Results are presented in Figures 3a-3d for relaxed DDs having a width of 100. In all the configurations tested, our approach provides a better upper bound than the `RAND`, `MIN`, `MPD` and `DEG` heuristics. For sparsest graphs (Figure 3a), the optimal solution is reached for almost all the instances. When the graphs are relatively sparse (Figure 3b), the linear relaxation provides the best bound. However, this trend decreases as the density of the graphs grows (Figures 3c and 3d). For these graphs, our model gives the best performance for all the instances. Results for restricted DDs with a width of 2 are depicted on Figures 4a-4d. Again, our model has the best results over 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 ($\nu = 16$).

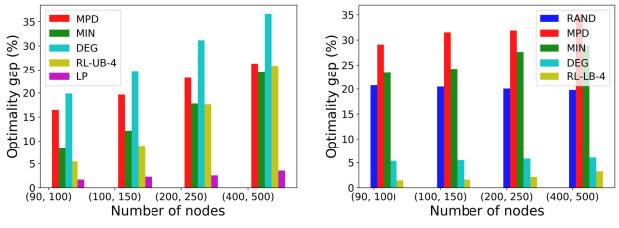
Analysis of Width Evolution Let us now consider the situation depicted in Figure 3b where RL−UB−4 provides a worse bound than the linear relaxation of the problem. Figure 5a depicts the evolution of the optimality gap when the model is tested on relaxed DDs of an increasingly larger width. As RAND provided results far outside the range of the other methods for relaxed DDs, we do not include it in the subsequent plots. The plot depicts that RL−UB−4 remains better than the other ordering heuristics tested, and when the DD width is sufficiently large ($w > 1000$) the LP relaxation bound is beaten and the optimal solution is almost reached ($w = 10000$). Figure 5b reports the execution time of the different methods. Concerning the RL model, only the time required for building the DD is reported. We do not report the training time on this experiment because it has to be amortized on all the graphs of the test set, which is dependent of the situation. The linear relaxation is the fastest method and is almost instantaneous. Concerning the orderings, RL−UB−4, MPD and DEG are static, and execution time for each generally increases similarly with the width, while MIN requires dynamically processing the nodes in a layer for determining the next vertex to insert.



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

Figure 5: Relaxed DDs of larger widths ($\nu = 4$).

Analysis of Graph Size Evolution In a similar way, this set of experiments aim to analyze how the learned models perform when larger graphs are considered. Results in Figure 6a depict the optimality gap of the different approaches for relaxed DDs ($w = 100$).

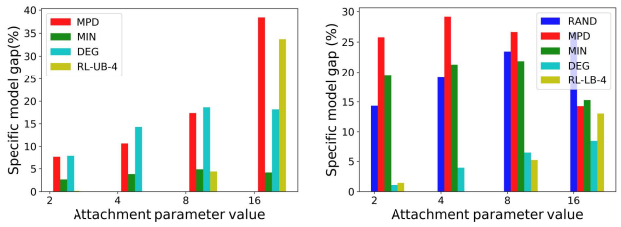


(a) Relaxed DDs ($w = 100$). (b) Restricted DDs ($w = 2$).

Figure 6: Relaxed/Restricted DDs for larger graphs ($\nu = 4$).

We can observe that the learned model remains robust against increases of the graph size although the gap between the other orderings progressively decreases. When the graph size is far beyond the size used for the training, the model strives to generalize which indicates that training on larger graphs should be required. The LP bounds for large graphs are out of range of DDs of this limited width. The same experiment is carried out for restricted DDs and reported in Figure 6b. Given that the optimality is reached even with small-width DDs, only a width of 2 is considered. Here, RL−LB−4 provides the best lower bound even for the largest graphs tested. This is consistent with other heuristics implemented through RL.

Performance on other Distributions This set of experiments aim to analyze the performance of the learned models when they are tested on a different distribution than that used for training. Figure 7 presents the relative gap with the model specifically trained on the distribution tested. For instance, when $\nu = 8$, the gap is computed using RL−UB−8 as reference (or using RL−LB−8 for restricted DDs). We use this measure instead of the optimality gap in order to nullify the impact of the instance difficulty. The gap is then null for the distribution used as reference (RL−UB−4 and RL−LB−4). Results show that the more the distribution is distant from the reference, the greater is the gap, which indicates that the learned model strives to generalize. For small perturbations ($\nu = 2$ and 8), good performances are still achieved. These results suggest that it is important to have clues on the distribution of the graphs that we want to access in order to feed appropriately the model during training.

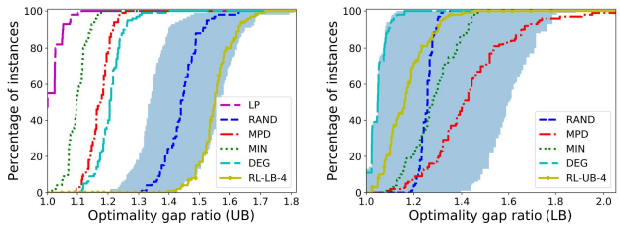


(a) Relaxed DDs ($w = 100$). (b) Restricted DDs ($w = 2$).

Figure 7: Performance on other distributions.

Analysis of the Ordering for Relaxed/Restricted DDs In our initial situation, we have considered a different model for learning the variable ordering related to relaxed and restricted DDs. Two separate models have then to be trained for getting the bounds of a single instance. An interesting question that may arise is the following: is a model trained using relaxed DDs can also be used for getting the ordering of restricted DDs or inversely ? The benefit is that only one model would be required for computing the bounds which will reduce consequently the training time. The applicability of this is the purpose of this set of experiments. Figure 8a replays the experiments of Figure 3b but using the model trained with restricted DDs (RL−LB−4) instead. Similarly, Figure 8b shows the situation of Figure 4b with RL−UB−4. As we can see, the models do not perform well in such cases and are similar or even worse than the random ordering. It indicates that both cases must be handled separately. On a

higher level of perspective, it empirically shows that an ordering which is efficient for one situation would not be irremediably good for the other one.



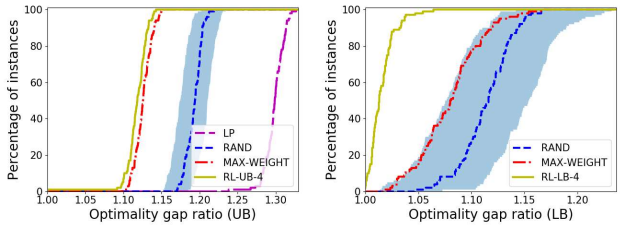
(a) Replay of Figure 3b. (b) Replay of Figure 4b.

Figure 8: Performance of the model trained with restricted DDs on relaxed DDs and inversely.

Experiments on the Maximum Cut Problem

Definition 2 (Maximum Cut Problem) *Let $G(V, E)$ be a simple undirected graph. A maximum cut of G is a subset of nodes $I \subseteq V$ such that $\sum_{(u,v)\in C} w(u, v)$ is maximized, where $C \subseteq E$ is the set of edges having a node in S and the other one in $V \setminus S$. The Maximum Cut Problem (MCP) is that of finding a maximum cut.*

As an example of its generalizability, our approach is also applied to the MCP. The DD is built according to formulation of (Bergman et al. 2016). The learning process and the model is the same as for the MISP. Generation of graphs is still done using a Barabasi-Albert distribution ($\nu = 4$) with edge weights uniformly and independently generated from $[1, 10]$. For training, weights are scaled with a factor of 0.01. The ordering obtained is compared with `RAND` and with the `MAX-WEIGHT` heuristic which selects the vertex having the highest sum of incoming weights (Bergman et al. 2016). The linear relaxation of a standard integer program of the MCP (Kahruman et al. 2007) is also considered. Results reporting the optimality gap of the three methods are presented in Figure 9 for relaxed ($w = 100$ for training and testing) and restricted DDs ($w = 2$).



(a) Relaxed DDs ($w = 100$). (b) Restricted DDs ($w = 2$).

Figure 9: Performance profiles for the MCP ($\nu = 4$).

In both cases, performances better than `RAND` and `MAX-WEIGHT` are reached, indicating that the learning is effective. Concerning relaxed DDs, the gap with `MAX-WEIGHT` is tighter, which could indicate that this heuristic already gives strong bounds and is then difficult to beat. Finally, the classical linear relaxation does not perform well on the MCP, which was already known (Avis and Umemoto 2003).

Conclusion

Objective function bounds are paramount to general and scalable algorithms for combinatorial optimization. Decision diagrams provide a novel and flexible mechanism for obtaining high-quality bounds whose output is amenable to improvement through machine learning, since the objective function bound obtained is directly linked to the heuristic choices taken. This paper provides a generic approach based on deep reinforcement learning for finding high-quality heuristics for variable orderings that are shown experimentally to tighten the bounds proven by approximate DDs. Experimental results indicated the promise of the approach when applied to the Maximum Independent Set Problem.

Insights from a thorough experimental evaluation indicate: (1) the approach generally outperforms variable ordering heuristics appearing in the literature; (2) the width chosen during training can have a negligible impact when applied to unseen instances; (3) the model generalizes well when the width is increased and, in most cases, is applied to larger graphs; (4) a separate model must be trained for relaxed and restricted DDs; (5) the approach generalizes to other problems, such as the Maximum Cut Problem; and (6) it is important to have a measure of the distribution on the evaluated graphs in order to be able to feed the model during training. This last point remains a challenge when extending the approach to real-world problems. As a future work, we plan to tackle it by generating new instances for training using generative models from the initial graphs. The idea is to augment the training set by generating new instances, that looks similar in structure to the initial instances, but that are still different.

To the best of our knowledge, this is the first paper to propose the use of machine learning in discrete optimization algorithms for the purpose of learning both primal and dual bounds in a unified framework. It opens new insights of research and multiple possibilities of future work, such as the application to different domains that utilize DDs as constraint programming, planning or verification of systems.

References

[Albert and Barabási 2002] Albert, R., and Barabási, A.-L. 2002. Statistical mechanics of complex networks. *Reviews of modern physics* 74(1):47.

[Arulkumaran et al. 2017] Arulkumaran, K.; Deisenroth, M. P.; Brundage, M.; and Bharath, A. A. 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

[Avis and Umemoto 2003] Avis, D., and Umemoto, J. 2003. Stronger linear programming relaxations of max-cut. *Mathematical Programming* 97(3):451–469.

[Bello et al. 2016] Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

[Bergman et al. 2012] Bergman, D.; Cire, A. A.; van Hoeve, W.-J.; and Hooker, J. N. 2012. Variable ordering for the application of BDDs to the maximum independent set problem. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, 34–49. Springer.

[Bergman et al. 2013] Bergman, D.; Cire, A. A.; van Hoeve, W.-J.; and Hooker, J. N. 2013. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing* 26(2):253–268.

[Bergman et al. 2016] Bergman, D.; Cire, A. A.; van Hoeve, W.-J.; and Hooker, J. 2016. *Decision diagrams for optimization*. Springer.

[Bergman, van Hoeve, and Hooker 2011] Bergman, D.; van Hoeve, W.-J.; and Hooker, J. N. 2011. Manipulating MDD relaxations for combinatorial optimization. In Achterberg, T., and Beck, J. C., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 20–35. Berlin, Heidelberg: Springer Berlin Heidelberg.

[Bollig and Wegener 1996] Bollig, B., and Wegener, I. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on computers* 45(9):993–1002.

[Bottou 2010] Bottou, L. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer. 177–186.

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

[Carbin 2006] Carbin, M. 2006. Learning effective BDD variable orders for BDD-based program analysis.

[Dai, Dai, and Song 2016] Dai, H.; Dai, B.; and Song, L. 2016. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, 2702–2711.

[Deudon et al. 2018] Deudon, M.; Cournut, P.; Lacoste, A.; Adulyasak, Y.; and Rousseau, L.-M. 2018. Learning heuristics for the TSP by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 170–181. Springer.

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

[Grumberg, Livne, and Markovitch 2011] Grumberg, O.; Livne, S.; and Markovitch, S. 2011. Learning to order BDD variables in verification. *CoRR* abs/1107.0020.

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

[Henderson et al. 2017] Henderson, P.; Islam, R.; Bachman, P.; Pineau, J.; Precup, D.; and Meger, D. 2017. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*.

[Kahruman et al. 2007] Kahruman, S.; Kolotoglu, E.; Butenko, S.; and Hicks, I. V. 2007. On greedy construction heuristics for the MAX-CUT problem. *International Journal of Computational Science and Engineering* 3(3):211–218.

[Khalil et al. 2017] Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6351–6361.

[Kingma and Ba 2014] Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

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

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

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

[Mnih et al. 2013] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

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

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

[Rumelhart, Hinton, and Williams 1986] Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning representations by back-propagating errors. *nature* 323(6088):533.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; 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.

[Sutton and Barto 1998] Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

[Watkins and Dayan 1992] Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.