Improving Branch-and-Bound Using Decision Diagrams and Reinforcement Learning

Augustin Parjadis[1(✉)], Quentin Cappart[1], Louis-Martin Rousseau[1], and David Bergman[2]

[1] École Polytechnique de Montréal, Montreal, Canada
{augustin.parjadis-de-lariviere,quentin.cappart, louis-martin.rousseau}@polymtl.ca
[2] University of Connecticut, Storrs, CT 06260, USA
david.bergman@uconn.edu

Abstract. Combinatorial optimization has found applications in numerous fields, from transportation to scheduling and planning. The goal is to find an optimal solution among a finite set of possibilities. Most exact approaches use relaxations to derive bounds on the objective function, which are embedded within a branch-and-bound algorithm. Decision diagrams provide a new approach for obtaining bounds that, in some cases, can be significantly better than those obtained with a standard linear programming relaxation. However, it is known that the quality of the bounds achieved through this bounding method depends on the ordering of variables considered for building the diagram. Recently, a deep reinforcement learning approach was proposed to compute a high-quality variable ordering. The bounds obtained exhibited improvements, but the mechanism proposed was not embedded in a branch-and-bound solver. This paper proposes to integrate learned optimization bounds inside a branch-and-bound solver, through the combination of reinforcement learning and decision diagrams. The results obtained show that the bounds can reduce the tree search size by a factor of at least three on the maximum independent set problem.

Keywords: Decision diagrams · Branch-and-bound · Reinforcement learning.

1 Introduction

Historically introduced for encoding Boolean functions and used for circuit design and verification [10,21], *Decision Diagrams* (DDs) have recently been reapplied in the field of combinatorial optimization [2,8,17], for example to sequencing problems [13] or the multidimensional bin packing problem [18]. Assuming a maximization objective, the optimal solution can be obtained in polynomial time in respect to the size of the decision diagram by following the

© Springer Nature Switzerland AG 2021
P. J. Stuckey (Ed.): CPAIOR 2021, LNCS 12735, pp. 446–455, 2021.
https://doi.org/10.1007/978-3-030-78230-6_28

longest path from the root to the terminal node of an exact DD. However, the size of exact DDs grows exponentially with the number of variables, which make them unsuitable for solving large problems. Recently, decision diagrams provided new means of obtaining bounds for combinatorial optimization problems that can be significantly better than those obtained via a traditional linear programming relaxation [6]. Bergman et al. [9] proposed to use DDs to encode a parametrizable and tractable approximation of the solution set. Such structures are referred to as *approximate* DDs.

The performance of this procedure highly depends on the ordering of the variables used to create the DDs. Better ordering can lead to tighter optimization bounds, which results in fewer nodes explored during the BnB search and an expected solution time reduction. Nevertheless, finding an ordering that yields the best bound is NP-hard and difficult to model. Typically heuristics have been considered for defining variable ordering [5] and only a few limited studies have proposed exact approaches for specific problem classes (see, e.g. [4]). Recently, Cappart et al. [12] proposed a deep reinforcement learning (DRL) [3,20] approach for computing the variable ordering. The idea is to train an agent to build a DD, with the incentive (i.e., the reward) to obtain bounds as tight as possible. However, this procedure has not been integrated in a BnB algorithm.

The contribution this paper makes is to illustrate the benefits and challenges of using both primal and dual bounds obtained with DRL-guided DDs within a BnB procedure. As a first experiment we apply this algorithm to the *maximum independent set problem* (MISP). Our preliminary results show that the proposed approach is able to prove optimality with significantly fewer nodes explored, due to the better bounds obtained. However, it is done at the expense of a significant increase in the execution time, as a deep neural network has to be called multiple times at each BnB node. An improved algorithm which uses caching and restricts the use of the DRL agent is also presented, and proves efficient on larger problems.

This paper introduces some preliminaries for DDs and reinforcement learning before presenting the BnB algorithm, followed by experimental results. Finally, the limitations of the current approach and directions for future research are discussed.

2 Learning Bounds Inside Branch-and-Bound

2.1 Decision Diagram-Based Branch-and-Bound

The DDs used in this work are Binary Decision Diagrams (BDDs), although the generalization to multi-valued decision diagrams is immediate. A BDD $B = (U, A, d)$ is a directed acyclic multi-graph in which the node set U is partitioned into layers $L_1, \ldots, L_{n+1}$. The set of solutions for a problem can be represented by a BDD in which each path from the root node r to the terminal node t encodes a feasible solution: each arc along the path gives the value of the variable associated with the layer the arc starts from, and a longest path in the diagram gives an optimal solution to the problem. Conversely, each feasible solution can be found

in the BDD. Each layer $L_1, \ldots, L_n$ is associated with a unique variable x_k of the problem with $k \in \{1, \ldots, n\}$. Each arc $a \in A$ goes from one layer to the next and has label $d(a) \in \{0, 1\}$ that encodes the values of the layer's associated binary variable.

For larger problems in combinatorial optimization, BnB algorithms are widely used to generate a search tree of manageable size using optimization bounds. Such bounds can be obtained via feasible solutions and linear relaxation, and we focus here on how approximated DDs provide a simple alternative for computing bounds. An exact DD can be *relaxed* by merging nodes of a layer to narrow the diagram without removing any solutions, but adding unfeasible solutions [9]. A relaxed DD provides a dual bound when solved, the quality of which depends on the amount of merging operations done. Likewise, a *restricted* DD can be created by removing nodes of an exact DD; some solutions are deleted but none are created, yielding a subset of solutions that can provide a primal bound. A branching process can also be conducted on DD nodes, which provides a complete BnB scheme based on DDs as an alternative to the classic linear programming-based BnB. A DD-based BnB algorithm is described in Algorithm 2. Detailed information is proposed by Bergman et al. [5].

2.2 Variable Ordering and Reinforcement Learning

Recently, a machine learning approach has been proposed to address the NP-hard problem of variable ordering for DDs [12] using reinforcement learning (RL) [24]. The goal of a RL agent is to maximize the expected sum of rewards obtained by learning a behavior policy. In the case of DD construction, the agent incrementally builds an approximated DD and observes the bound given by the partially built diagram. The rewards given as feedback are defined by the evolution of the bound obtained (a reduction of an upper bound is encouraged, whereas an increase is penalized and vice versa for a lower bound). The state observed by the agent is a function of the problem instances and of the variables already added in the current DD. An action consists in selecting a new variable to add to the next layer of the DD. Finding a policy maximizing the action-value function for all states and actions is hard and a practical solution is to compute an approximation of Q using a Q-learning algorithm [25] applied to a graph neural network. The exact procedure can be found in can be found in [12].

2.3 The Branch-and-Bound Algorithm with a RL Agent

This section presents the main contribution of the paper: how to implement a BnB algorithm using DDs together with RL. The process is as follows. At each node explored during a DD-based BnB algorithm, a relaxed and a restricted DD are built and an ordering for the variables left at this stage of the search has to be determined for each DD. Currently, heuristics that try to greedily limit the width of the DD are used, and perform much better than simple lexicographic or random orderings.

We propose to work on the ordering of the approximate DDs by using a RL agent on a graph embedding rather than handcrafted heuristics. Two steps are to be considered for building this agent: (1) a training phase, consisting of learning a good policy for the problem at hand, and (2) a solving phase, corresponding to the execution of the BnB algorithm.

Vectorized Representation. A graph neural network (GNN) [11,19] is used to obtain a vectorized representation of a graph by computing node embeddings. The vectorization is built as follows: (1) the problem instance is represented as a graph (e.g., a MISP instance), (2) each vertex/edge of the nodes are decorated with relevant features (e.g. the weight of each vertex), (3) A GNN is used to obtain an d-dimensional embedding for each vertex of the graph, and (4) the embedding is given as input to a fully-connected neural network in order to obtain the final Q-values that are used for the prediction, indicating the quality of each vertex to be inserted in the current embedding.

Training Phase. The training is based on neural fitted Q-learning with a set of randomly generated graphs, and returns the weights $\mathbf{w}$ for the approximated action-value function $\hat{Q}$. The Q-value approximation is obtained with the use of a GNN with the library `structureToVec` [14]. The standard Q-learning algorithm can be enriched in several classic ways, among which are mini-batches to guarantee a better gradient descent based on several examples instead of one, reward scaling for a better handling of large reward quantities and an adaptive ϵ-greedy policy to move away from a local optimum, balancing exploration and exploitation. Details on the learning phase can be found in [12]. As the training uses the bound obtained via the partially built DD, the RL agent is dependent on the type of bound used. Therefore we consider two agents, one trained for relaxed DDs for which a low value upper bound is desired, and the other for restricted DDs for which the highest possible lower bound is desired. Once the parameters $\mathbf{w}$ for the Q-value have been learned, a simple policy can be derived as described below.

Solving Phase. The solver uses a trained RL agent each time a DD is developed. Algorithm 1 describes how an ordering is obtained by calling the agent several times, and Algorithm 2 describes the BnB algorithm using the DDs constructed with the RL agents. New variable orderings are computed at each node of the BnB to build the restricted and relaxed DDs. The RL agent is designed to obtain high-quality bounds, which then allows to update the lower bound or prune the node if applicable. Let B_{ut} be the DD induced by all nodes and paths going from u to t (which gives in particular $B_{rt} = B$). If B_{ut} is an exact DD, B'_{ut} designates a restriction and $\overline{B}_{ut}$ a relaxation of B_{ut}. If a restricted DD B'_{ut} is exact, no restriction was needed and thus no branching is performed. The variable orderings are obtained as described below.

In Algorithm 1, the state of the problem is given to the agent that responds with the Q-values for each variable. The variable u that maximizes the Q-value

Algorithm 1. DD construction with RL agent

1: $B = (\{L_1, \ldots, L_n\}, A, d)$ an empty DD, to be built with the variables V
2: **for** $j \in \{1, n\}$ **do**
3: $u = \text{argmax}_{a \in V} \hat{Q}(s, a, \mathbf{w})$
4: *build layer* L_j *with the variable* u
5: $V \leftarrow V \backslash \{u\}$, $b_u = 1$
6: **end for**
7: return B

learned by the GNN is chosen and added to the partially built DD. A feature b_u for each $u \in V$ indicates if a variable has already been selected for the DD construction. After the selection of the variable u, the feature is updated. This modifies the state of the problem, so the Q-values are computed again to select the next variable. The process continues until all the variables have been considered and the DD is complete.

Algorithm 2. DD based branch-and-bound, 2 RL agents

1: $z_{\text{opt}} = -\infty$, $\mathcal{Q} = \{r\}$
2: **while** $\mathcal{Q} \neq \emptyset$ **do**
3: select node $u \in \mathcal{Q}$
4: create restricted DD B'_{ut} with Algorithm 1
5: **if** $v^*(B'_{ut}) > z_{\text{opt}}$ **then** $z_{\text{opt}} = v^*(B'_{ut})$ **end if**
6: **if** B'_{ut} non exact **then**
7: create relaxed DD $\overline{B_{ut}}$ with Algorithm 1
8: if not pruned, select exact cutset to add to $\mathcal{Q}$
9: **end if**
10: **end while**

Complexity. Obtaining a variable ordering by calling a GNN is expensive because of the nature of the steps involved. The node embeddings are created by a message passing algorithm and fed to a fully connected neural network. This has to be done for each variable in the ordering, resulting in a time complexity of $\mathcal{O}(|E| \times n^2)$. In the next section, Algorithm 1 for variable ordering is compared against a classic heuristic called *MinState*, which has a time complexity of $\mathcal{O}(w \times n)$; with w the width of the DD. The additional complexity brought by the GNN should thus have to be justified with an efficient node reduction during the BnB.

3 Experimental Results

The goal of the first set of experiments is to show that a significant reduction of nodes explored in the tree search is possible with a well trained GNN. However, it is done at the expense of the computation time. The second experiment shows

how this issue can be mitigated using a caching mechanism, and a hybrid heuristic. Similar to previous works [7,12], the case study considered is the maximum independent set problem (MISP). The solver is written in `C++` and is implemented upon the code of Cappart et al. [12] for the RL agent part and Bergman et al. [5] for the DD-based BnB. The learning and the solving were ran on a Dell XPS 15 9570 with a Intel i7-8750H 2.20 GHz CPU. The training time was limited to eight hours. All instances considered have been randomly generated using a Barabasi-Albert scheme with the density parameter $\nu = 4$ [1].

Definition 1 (Maximum Independent Set Problem). *An independent set I in a weighted graph $G = (V, E)$ is a subset $I \subseteq V$ in which 2 vertices cannot be adjacent. The problem consists of finding an independent set of maximum weight, i.e. maximizing the function $f(x) = \sum_{k=1}^{|V|} w_k x_k$ such that $\forall (i,j) \in E, x_i + x_j \leq 1$ with x_k indicating whether the vertex k is selected or not, and w_k being the weight associated to this vertex.*

3.1 Performances of the Learned Variable Ordering

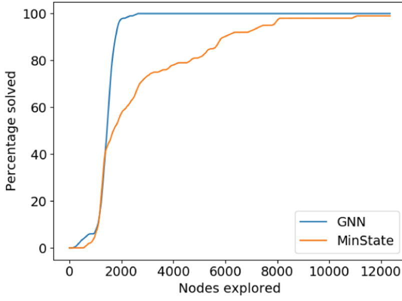
We plot in Fig. 1 the number of nodes explored during the DD-based BnB over 100 random instances. For the variable orderings, the *GNN* algorithm uses a GNN trained for relaxation over random graphs with the same distribution. Restriction is still done with a MinState ordering. The GNN consists of 4 iterations of belief propagation and a neural network with 2 layers of size 64. The *heuristic* algorithm uses MinState for relaxation and restriction. The results can be found in Table 1, with the average time in seconds. We can see that even if the number of nodes is drastically reduced, the execution time remains an important concern compared to MinState. Figure 1 shows the percentage of the graphs solved under a given number of nodes in performance profile [22]: the higher the curve, the better. Overall, the number of nodes needed to solve all the MISP instances is typically reduced by a factor of three to four with the RL agent for graphs over 100 nodes, noting that 90% of the instances were solved in fewer nodes.

Table 1. Average time and nodes explored for 100 MISP instances of size n

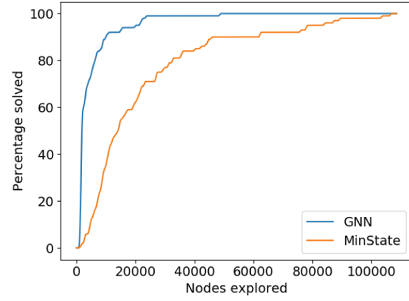
Algorithm	$n = 80$		$n = 100$		$n = 120$	
	Nodes	Time	Nodes	Time	Nodes	Time
GNN	1416	16.70	4628	79.90	34434	878.10
GNN+	2131	2.40	13152	10.00	81015	41.10
MinState	2802	0.32	22680	5.30	100822	40.40

3.2 Caching to Save Computation Time

To avoid calling the agent too often and speed-up the resolution, the GNN-computed orderings are stored and re-used for subsequent DDs that might have



(a) n = 80



(b) n = 100

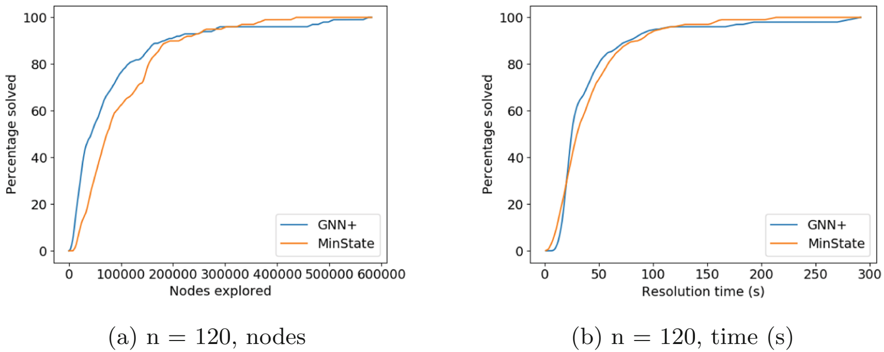
Fig. 1. Performance profile of a GNN agent and MinState for variable ordering over 100 instances of the MISP

a similar structure. To re-use an ordering, the number of variables for the current DD has to be close to the number of variables of the stored ordering and be included in it. When the number of variable differs too much, the stored ordering can be quite poor for the current DD (similar to a random ordering). This sacrifices some precision in favor of computational efficiency. Furthermore, the computation time bottleneck is expressed the most when it is needed the least, which is at the end of the BnB tree for small DDs. Bound improvements there are small and do not have major consequences, and small DDs are very fast to build with little need for a GNN. We experiment on the use of the RL agent before a given threshold for the number of nodes (here, 100 BnB nodes), and the use of the MinState heuristic after this threshold to close the search faster. Those improvements are natural given the cost of a GNN feed-forward operation, and we observe in Table 1 and in Fig. 2 with the algorithm GNN+ that the average number of nodes explored during a search can in fact be effectively reduced while keeping a competitive solving time for large problems.

3.3 Discussion

As highlighted in the experiment, the learned bounds can be successfully leveraged in the solver in order to reduce the number of nodes explored. However, the execution time is an important bottleneck, which makes the hybridization challenging to design, and additional mechanisms should be considered in order to improve the efficiency. *Caching* is one of them but is still not enough. We made other attempts to get improvements, such as (1) using the same agent for the restriction and the relaxation, (2) using a fixed ordering during the complete BnB process instead of dynamically recomputing it, (3) calling the GNN only every n steps or at random intervals, but without much success. We believe that finding other ways to improve is a promising research direction. A similar question has been addressed by Gupta et al. [16] for standard MIP solvers. They show that expressive and costly GNNs can be combined with inexpressive

but cheap fully connected neural networks in order to obtain a better trade-off between prediction quality and computational efficiency.



(a) n = 120, nodes

(b) n = 120, time (s)

Fig. 2. Performance profile of an improved GNN agent and MinState for variable ordering over 100 instances of the MISP

4 Conclusion

DDs provide a new flexible and general methodology for generating tight bounds for optimization problems, and DD-based BnB algorithms are often competitive with recent integer programming solvers. This paper presents a method that takes advantages of the flexibility of DDs for bound generation in BnB through machine learning. The reduction of the number of nodes indicates that this is a promising research direction, but subject to important challenges (e.g. the execution time). The generic nature of DD construction and BnB solving for discrete optimization problems makes for an interesting combination, with possible application to other problem classes like the maximum cut problem [6], the set covering problem [9] and the traveling salesperson problem [15,23].

The next step to better bridge the gap between machine learning and optimization would involve a general framework that takes advantage of the graph structure and recursive formulation of a problem by learning to construct DDs with high quality bounds for the problem, and solving it with an efficient BnB algorithm. Data generation for the training step would remain a challenge.

The use of machine learning to generate both lower and upper bounds for combinatorial optimization proved efficient and opens the door to multiple possibilities for future research on the integration of these disciplines. Applying deep learning tools to optimization faces practical bottlenecks that have to be overcome for practical applications and broad adoption in exact optimization solvers.

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. **74**, 47–97 (2002)
2. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 118–132. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_11
3. Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: A brief survey of deep reinforcement learning. CoRR abs/1708.05866 (2017). http://arxiv.org/abs/1708.05866
4. Behle, M.: On threshold BDDs and the optimal variable ordering problem. In: Dress, A., Xu, Y., Zhu, B. (eds.) COCOA 2007. LNCS, vol. 4616, pp. 124–135. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73556-4_15
5. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS J. Comput. **28**(1), 47–66 (2016)
6. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.: Decision Diagrams for Optimization. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42849-9
7. Bergman, D., Cire, A.A., van Hoeve, W.-J., Hooker, J.N.: Variable ordering for the application of BDDs to the maximum independent set problem. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 34–49. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29828-8_3
8. Bergman, D., Cire, A.A., van Hoeve, W.J., Yunes, T.: BDD-based heuristics for binary optimization. J. Heuristics **20**(2), 211–234 (2014). https://doi.org/10.1007/s10732-014-9238-1
9. Bergman, D., van Hoeve, W.-J., Hooker, J.N.: Manipulating MDD relaxations for combinatorial optimization. In: Achterberg, T., Beck, J.C. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 20–35. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21311-3_5
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **100**(8), 677–691 (1986)
11. Cappart, Q., Chételat, D., Khalil, E., Lodi, A., Morris, C., Veličković, P.: Combinatorial optimization and reasoning with graph neural networks. arXiv preprint arXiv:2102.09544 (2021)
12. Cappart, Q., Goutierre, E., Bergman, D., Rousseau, L.M.: Improving optimization bounds using machine learning: decision diagrams meet deep reinforcement learning. Proc. AAAI Conf. Artif. Intell. **33**, 1443–1451 (2019)
13. Cire, A.A., van Hoeve, W.J.: Multivalued decision diagrams for sequencing problems. Oper. Res. **61**(6), 1411–1428 (2013). https://doi.org/10.1287/opre.2013.1221
14. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data. In: International Conference on Machine Learning, pp. 2702–2711 (2016)
15. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.-M.: Learning heuristics for the TSP by policy gradient. In: van Hoeve, W.-J. (ed.) CPAIOR 2018. LNCS, vol. 10848, pp. 170–181. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93031-2_12
16. Gupta, P., Gasse, M., Khalil, E., Mudigonda, P., Lodi, A., Bengio, Y.: Hybrid models for learning to branch. In: Advances in Neural Information Processing Systems, vol. 33 (2020)

17. Hadzic, T., Hooker, J.: Postoptimality analysis for integer programming using binary decision diagrams. In: GICOLAG Workshop (Global Optimization), Vienna. Technical report, Carnegie Mellon University (2006)
18. Kell, B., van Hoeve, W.-J.: An MDD approach to multidimensional bin packing. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 128–143. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_9
19. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
20. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**, 436–44 (2015). https://doi.org/10.1038/nature14539
21. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Syst. Tech. J. **38**(4), 985–999 (1959)
22. Moré, J.J., Dolan, E.D.: Benchmarking optimization software with performance profiles. Math. Program. **91**, 201–213 (2002). https://doi.org/10.1007/s101070100263
23. O'Neil, R.J., Hoffman, K.: Decision diagrams for solving traveling salesman problems with pickup and delivery in real time. Oper. Res. Lett. **47**(3), 197–201 (2019)
24. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
25. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992). https://doi.org/10.1007/BF00992698