Using Clustering to Strengthen Decision Diagram Bounds for Discrete Optimization

Mohsen Nafar, Michael Römer

Management Science and Business Analytics department, Bielefeld University
mohsen.nafar@uni-bielefeld.de, michael.roemer@uni-bielefeld.de

Abstract

Offering a generic approach to obtaining both upper and lower bounds, decision diagrams (DDs) are becoming an increasingly important tool for solving discrete optimization problems. In particular, they provide a powerful and often complementary alternative to other well-known generic bounding mechanisms such as the LP relaxation. A standard approach to employ DDs for discrete optimization is to formulate the problem as a Dynamic Program and use that formulation to compile a DD top-down in a layer-by-layer fashion. To limit the size of the resulting DD and to obtain bounds, one typically imposes a maximum width for each layer which is then enforced by either merging nodes (resulting in a so-called relaxed DD that provides a dual bound) or by dropping nodes (resulting in a so-called restricted DD that provides a primal bound). The quality of the DD bounds obtained from this top-down compilation process heavily depends on the heuristics used for the selection of the nodes to merge or drop. While it is sometimes possible to engineer problem-specific heuristics for this selection problem, the most generic approach relies on sorting the layer's nodes based on objective function information. In this paper, we propose a generic and problem-agnostic approach that relies on clustering nodes based on the state information associated with each node. In a set of computational experiments with different knapsack and scheduling problems, we show that our approach generally outperforms the classical generic approach, and often achieves drastically better bounds both with respect to the size of the DD and the time used for compiling the DD.

Introduction

Solving discrete optimization problems is a challenging task which has kept busy generations of researchers from various fields such as Mathematics, Computer Science and and Operations Research. Given the impressive progress in the field of Artificial Intelligence (AI) and Machine Learning (ML) in the recent years, it seems natural that there is an ever-increasing amount of research that aims at leveraging the power of ML for solving optimization approaches, see e.g. the surveys (Bengio, Lodi, and Prouvost 2021; Kotary et al. 2021; Cappart et al. 2023).

The vast majority of the research dedicated to employing ML approaches for discrete optimization either deals with speeding up exact optimization approaches, e.g. by learning to take better branching decisions in branch-and-bound solvers (Khalil et al. 2016), or with using ML to replace human-designed algorithmic decisions within heuristic solution approaches. As an example, the recent years have seen tremendous improvements in so-called Deep Reinforcement Learning (DRL) approaches for routing problems which nowadays come very close to the best handcrafted heuristics that build upon decades of research (Hottung, Kwon, and Tierney 2021). While ML-based heuristics provide high-quality primal bounds for discrete optimization problems, there are relatively few works focusing on exploiting ML techniques for obtaining strong dual bounds, despite the fact that dual bounding mechanisms constitute a critical ingredient in exact discrete optimization solvers.

Perhaps one of the first works aiming at strengthening dual bounds with the help of ML was (Cappart et al. 2019) who proposed to employ DRL to improve bounds obtained with so-called approximate Decision Diagrams (DDs). DDs, initially introduced for representing boolean circuits, are layered graphical data structures that can be used for compactly representing the solution space of a discrete optimization problem. In particular, one can construct two types of limited-size approximate DDs that provide optimization bounds: Restricted DDs in which certain feasible nodes are discarded in order to obtain an under-approximation of the solution space and relaxed DDs in which certain nodes are merged in order to obtain an over-approximation that provides a dual bound. As demonstrated by Bergman et al. (2016), these bounds can be used in a purely DD-based branch-and-bound algorithm that is able to achieve state-of-the art performance for certain discrete optimization problems. In addition to their computational efficiency for certain problem types, one of the key benefits of DD-based approaches is that they are highly generic in the sense that in order to apply them, one basically only needs a Dynamic Programming (DP) formulation of the problem to be considered along with the specification of a so-called merge operator. For an excellent survey of the recent advancements in DD-based approaches for solving discrete optimization problems, we refer to (Castro, Cire, and Beck 2022).

The quality of the bounds obtained with approximate DDs

depends on certain heuristic decisions to be taken during the DD compilation process. It thus seems natural to harness the power of modern ML approaches for guiding those decisions. Actually, in the above-mentioned paper, Cappart et al. (2019) use DRL to determine the so-called variable ordering, that is, the order in which decision variables are considered when compiling a DD layer by layer in a top-down fashion. They show that for a given maximum width of each layer, the ML-supported approach can substantially improve the bounds compared to the variable ordering heuristics considered in the literature. In two follow-up works (Parjadis et al. 2021; Cappart et al. 2022), the authors show that despite the fact that the ML-based compilation of approximate DDs is slower than the standard approaches, this bound improvement leads to a significant overall speed-up of an exact DD-based branch-and-bound solver.

In this paper, we follow this line of research by using ML to guide taking another critical decision when compiling approximate DDs: The decision of which nodes to merge (in case of a relaxed DD) or to discard (in case of a restricted DD) in the case that the number of nodes in a DD layer exceeds the maximum permitted width. For the rest of this paper, we refer to this decision as *node selection*. Given the importance of this decision for the quality of the bounds, there has been considerable research on devising good node selection heuristics. These heuristics, which will be reviewed in more detail in the next section, rely on the information associated with each node and typically either rely on sorting the nodes according to some criterion and selecting the "worst" for discarding / merging, or on based similarity of nodes. While some existing node selection heuristics are highly generic (e.g. since they only rely on objective function information), most of the heuristics considered in the literature are tailored to the problem under consideration.

The main contribution of this paper is to propose a new ML-based node selection heuristic that relies on clustering the nodes according to the state information associated with each node during the compilation process. One of the advantages of this approach is that it is highly generic: it does not require specifying any problem-dependent heuristic or strategy since it operates with feature information that can be inferred from the DP formulation of the problem under consideration. In a set of computational experiments with five different discrete optimization problems, our approach consistently outperforms the standard generic node selection approach from the DD literature, often achieving drastically better bounds both with respect to the size of the approximate DD and the time used for its compilation.

Also using clustering within DD-based combinatorial optimization, the work most closely related to ours is (Coppé, Gillard, and Schaus 2023). The authors use state clustering to obtain an aggregated DP problem that is exactly solved, and the results are used to guide their DD-based Branch-and-Bound algorithm in various ways. In particular, they use the aggregate solution to assign scores to sort nodes in the node selection problem, which is very different from our approach that directly uses the node clusters for merging (relaxed DD) and for selecting one candidate per cluster to be kept (restricted DD).

Decision Diagrams for Optimization

A decision diagram $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ is a layered directed acyclic graph with node set $\mathcal{N}$ and arc set $\mathcal{A}$. The paths in $\mathcal{D}$ represent solutions to a discrete optimization problem $\mathscr{P}$ with a maximization objective function f and an n-dimensional vector of decision variables $x_1, \ldots, x_n \in \mathbb{Z}$. The node set $\mathcal{N}$ is partitioned into $n+1$ layers $\mathcal{N}_1, \ldots, \mathcal{N}_{n+1}$, where $\mathcal{N}_1 = \{\mathbf{r}\}$ and $\mathcal{N}_{n+1} = \{\mathbf{t}\}$ for a *root node* $\mathbf{r}$ and a *terminal node* $\mathbf{t}$. Each arc $a = (u, v)$ connects two consecutive layers, and is associated with a decision $d(a)$ representing the assignment $x_u = d(a)$. This means that a path $p = (a_1, \ldots, a_n)$ starting from $\mathbf{r}$ and ending at $\mathbf{t}$ represents the solution $x(p) = (d(a_1), \ldots, d(a_n))$. We denote the set of all $\mathbf{r}$-$\mathbf{t}$ paths with $\mathcal{P}$, and we refer to the solutions to $\mathscr{P}$ represented by $\mathcal{P}$ with $\mathrm{Sol}(\mathcal{D})$. Moreover, each arc a has length $\ell(a)$ and $\sum_{i=1}^{n} \ell(a_i)$ provides the length $\ell(p)$ of path p.

We refer to $\mathcal{D}$ as exact if $\mathrm{Sol}(\mathcal{D}) = \mathrm{Sol}(\mathscr{P})$ and for each path $p \in \mathcal{P}$ we have $\ell(p) = f(x(p))$. In that case, one can determine an optimal solution to $\mathscr{P}$ by determining longest $\mathbf{r}$-$\mathbf{t}$ path in $\mathcal{D}$. An aspect limiting the practical usefulness of exact DDs is their size which in general is exponential in the number of variables n. To address this issue, one can resort to smaller approximate DDs that can be used to obtain upper or lower bounds for the solutions of $\mathscr{P}$, and that can be used e.g. in a DD-based branch-and-bound procedure.

There are two types of approximate DDs: In a *restricted* DD $\mathcal{D}$, one aims at considering only promising nodes and arcs, meaning that $\mathrm{Sol}(\mathcal{D}) \subseteq \mathrm{Sol}(\mathscr{P})$, that is, not all feasible solutions to $\mathscr{P}$ are represented as paths in $\mathcal{D}$, and thus, the longest path in a restricted DD provides a lower bound to $\mathscr{P}$. The second type of approximate DD, the *relaxed* DD, provides an upper bound: In a relaxed DD, we have $\mathrm{Sol}(\mathcal{D}) \supseteq \mathrm{Sol}(\mathscr{P})$, that is, the set of paths may contain paths associated with infeasible solutions to $\mathscr{P}$. Regarding the objective function value, every path a relaxed DD needs to satisfy $\ell(p) \geq f(x(p))$, that is, the length of a path in a relaxed DD is assumed not to underestimate the true objective function of its associated solution in $\mathscr{P}$. In both restricted and relaxed DDs, a common approach to control the size of the DD is to impose a maximum width W for each layer.

As explored in detail in (Hooker 2013), DDs exhibit a close link to Dynamic Programming (DP): From a DP perspective, an exact DD for a discrete optimization problem $\mathscr{P}$ is very similar to the state-transition graph of a DP formulation of $\mathscr{P}$ in which every node u is associated with a state S_u and every arc a is associated with a state transition induced by the decision $d(a)$ associated with a. S_u is an element of the state space $\mathcal{S}$; the state $S_{\mathbf{r}}$ associated with the root node $\mathbf{r}$ is the so-called initial state. The state S_v of the target node v of the arc depends on the state S_u of the arc's source node as well as on d and is computed by the state-transition function $f(S_u, d)$. The (possibly state-dependent) objective function contribution of a decision is computed by a reward function $g(S_u, d)$. Finally, the set of out-arcs of a node u is determined by the set feasible decisions $X(S_u)$ given state S_u.

Given a DP formulation DP comprising the definition of the state space $\mathcal{S}$ including the initial state $S_{\mathbf{r}}$, the func-

tions X, f and g, one can compile a decision diagram $\mathcal{D}$ using a top-down compilation algorithm which is akin to a forward dynamic programming algorithm which is sometimes referred to as DP by reaching (Kellerer, Pferschy, and Pisinger 2004). A variant of such a top-down compilation procedure is displayed in Algorithm 1. The procedure takes a DP formulation DP, a DD $\mathcal{D}$ containing only the root node and the maximum width W. Calling the algorithm with an unlimited width W will yield an exact DD and depending on the operation performed in line 8, it will result in a restricted or relaxed DD.

Algorithm 1: Generic Top-Down compilation algorithm

1: **procedure** COMPILETOPDOWN(DP, $\mathcal{D}$, W)
2: **for** $j = 1$ to n **do**
3: **for all** $u \in \mathcal{N}_j$ **do**
4: **for all** $d \in X(S_u)$ **do**
5: $v =$ GETORADDNODE($\mathcal{N}_{j+1}, f(S_u, d)$)
6: ADDARC(u,v,d)
7: **if** $|\mathcal{N}_{j+1}| > W$ **then**
8: RELAXLAYER($\mathcal{N}_{j+1}$) or RESTRICTLAYER($\mathcal{N}_{j+1}$)

The algorithm proceeds layer by layer, starting from the root and ending at layer n which is the last layer before the terminal node $\mathbf{t}$.

For each layer j, it iterates through all nodes $\mathcal{N}_j$. For each node u, it considers all feasible decisions, and for each feasible decision d, the function GETORADDNODE is used to determine the target node v in the next layer $j+1$ by either retrieving the node associated with the resulting state $f(S_u, d)$ if it already exists, or by creating a new one. If $j = n$, GETORADDNODE always returns the terminal node $\mathbf{t}$. The algorithm then adds an arc associated with decision d from u to the target node v. After the creation of all nodes in layer $j+1$, it is checked whether the number of nodes exceeds W. If that is the case, the size of the layer is reduced by discarding (in a restricted DD) or merging (in a relaxed DD) nodes.

For the case of compiling a relaxed DD, the reduction of the size of the layer is achieved by merging nodes, that is by redirecting the incoming arcs of nodes that are being merged to the merged node. This way, the partial paths ending at the merged nodes are still available. In order to ensure that no feasible completion of any of the merged nodes is lost, one requires a problem-specific merge operator $\oplus$ for the states associated with the two nodes, see (Hooker 2017) for a discussion of the conditions a valid merge operator needs to satisfy.

In any case, the node selection strategy, that is, the strategy to select which nodes to discard or merge is critical for the quality of the bounds, and thus, has been subject to some amount of research. One of the most popular strategies is to sort the nodes according to some criterion, to keep the $W-1$ most promising states and to discard (in case of a restricted DD) or merge (in case of relaxed DD) the remaining states. A standard and problem-agnostic criterion is to sort according to the objective function value of the partial path ending at each of the nodes. This approach, which we will use as a baseline approach later, will be referred to as sortObj in the rest of this paper. Other criteria being used for sorting the nodes take the state information associated with the nodes into account.

Example. Let us consider the problem of scheduling jobs on two identical machines with the objective of minimizing the total weighted job completion time. Using standard scheduling notation (see e.g. Graham et al. 1979), this problem can be written as $P_2||\sum w_j C_j$, and it was proven to be NP-hard (Bruno, Coffman Jr, and Sethi 1974) and (Lenstra, Kan, and Brucker 1977). An instance of this problem consists of n jobs each associated with a processing time p_i and a weight w_i.

To formulate this problem as a DP, we assume that each stage is associated with a job i. In each stage, the decision d_i corresponds to assigning job i to either machine 1 or to machine 2. Each state S can be represented as a 2-tuple (s_1, s_2) where s_m is the total processing time on machine m given a set of partial assignments, and the transition function f adds the processing time of job i to the state coordinate associated with the machine determined by the assignment decision. The merge operator $\oplus$ that is required for merging states in a relaxed decision diagram corresponds to the element-wise minimum of the state tuple.

In the following numerical example, we consider an instance with 4 jobs with the processing times and weights given by the vectors $p = [4, 2, 5, 6]$ and $w = [2, 3, 2, 2]$.

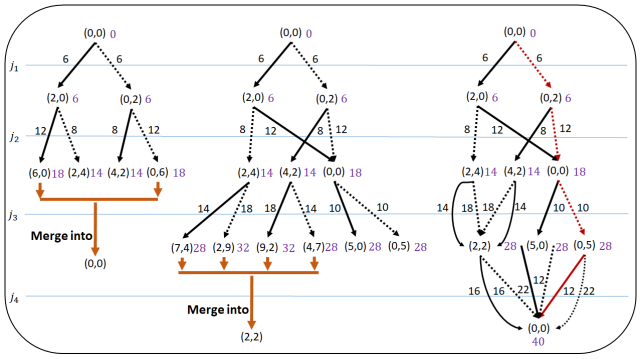


Figure 1: A relaxed DD compiled via sortObj for $P_2||\Sigma w_j C_j$ with $W = 3$ gives a solution with 16% gap.

Figure 1 illustrates the construction of a relaxed DD with a maximum width $W = 3$ in which the relaxation of each layer is performed according to the standard approach that is based on sorting the nodes according to their objective function values. Solid and dotted arcs show assignment of the job to machine 1 and 2, respectively. The label of each arc corresponds to the cost contribution (corresponding to the completion time of the job under consideration). The node labels correspond to the state tuple and the objective function value (displayed in purple) of the partial solution ending at each node. A shortest path from the root to the terminal node is highlighted in red, its length is 40, which corresponds to 83 % of the optimal value of 48 of the instance under consideration. □

One of the disadvantages of sorting-based node selection approaches is that creating a single merged node can lead to the fact that highly different nodes are merged which can negatively affect the achievable dual bounds. To address this issue, some authors aim at grouping nodes to merge according to some similarity measure. As an example, (Horn et al. 2021) propose to use so-called collector nodes that aim at merging states that have the same value with respect to a labeling function. A similar approach was recently used by (de Weerdt, Baart, and He 2021) who merge nodes based on partitioning the state space for a single machine scheduling problem with release times, deadlines, setup times and rejection. The key intuition behind these similarity-based approaches is that similar nodes should have similar set of feasible completions, and thus, the risk of negatively impacting the bounds by merging very heterogeneous nodes is smaller.

Clustering-Based Approximate DD Layers

The approach proposed in this paper can also be considered as a similarity-based approach to node selection. However, while the papers mentioned above rely on partitioning the state space, e.g. by devising a labeling function, we propose to obtain sets of similar nodes by applying standard clustering approaches to group the nodes in the layer. This approach has several advantages: First, the modeler does not need to specify any problem-specific labeling function or partitioning of the state space $\mathcal{S}$ since the clustering algorithms are problem-agnostic and use information that is readily available in any DP-based DD compilation method. Note, however, that it is nonetheless possible to adapt the clustering in different ways by specifying distance functions or selecting and tuning clustering algorithms. Second, the approaches mentioned above rely on an "a priori" partitioning of the full state space $\mathcal{S}$, which may lead to situations where similar states end up in different partitions. Since our clustering approach directly operates with the node states in the layer $\mathcal{N}_{j+1}$, we can assume that the grouping of the nodes is better adapted to concrete set of nodes under consideration. Third, and related to the second point, an a priori partitioning approach may lead to many "empty buckets", and thus one cannot directly control the number of nodes resulting from the partitioning. In our approach, we can control the size of the layer by employing clustering approaches such as k-means clustering in which we can directly control the width of the layer to be constructed.

Observe that our work is not the first to employ ML in the context of node selection decisions for compiling relaxed DDs: (Frohner and Raidl 2019) used a binary classification approach based on information from a limited lookahead for dynamically determining the merge heuristic to use in a given layer. While they show that this approach can achieve better bounds than using only a single merge heuristic, the reliance on the lookahead results in a large computational overhead compared to non-ML based approaches. In our approach, however, ML plays a much more direct role in supporting node selection since the result of the clustering algorithm can be directly mapped to the groups of nodes to be merged.

Our clustering approach to node selection can in principle work with almost any clustering algorithm that can operate with the state information defined in the DP model for the discrete optimization model under consideration. In this paper, and in our computational results, we resorted to a standard implementation of a general k-means clustering algorithm, allowing us to explicitly specify the number of clusters to be constructed. In case of relaxed DDs, the clustering-based version of RELAXLAYER proceeds by applying the clustering algorithm to all the associated nodes in the layer. Observe that this clustering is performed online, that is, without any pre-trained clustering model. For each of the k clusters, we apply the merge operation to obtain the merged state. In the case of a restricted DD, the same clustering logic applies. However, instead of applying the merge operator, we select the node with the best objective function value and discard the other nodes from the cluster.

Example (continued). We illustrate our clustering-based approach for compiling a relaxed DDs for the example introduced in the previous section. Now, instead of using the classical sorting-based approach, we cluster the nodes according to their states and merge all the nodes being present in the same cluster. Figure 2 displays the progression of the top-down compilation based on the clustering for the same instance that was used in Figure 1. Specifically, the clustering is obtained using the k-means clustering algorithm. Different clusters are highlighted with different colors.

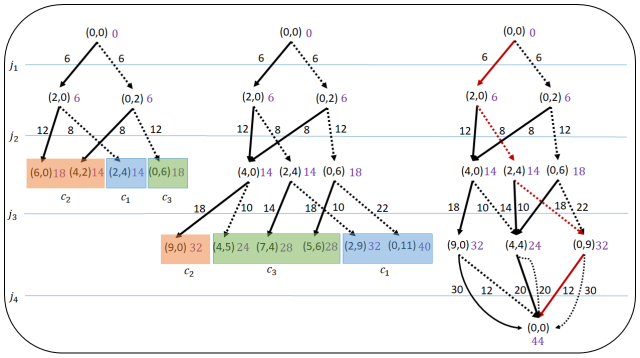


Figure 2: A relaxed DD compiled via clustering-based approach for $P_2||\Sigma w_j C_j$ with $W = k = 3$ gives a solution with 8% gap.

It turns out that the clustering leads to a different configuration of the layers that need to be relaxed, and that the bound is much stronger than the bound obtained with the standard "sortObj" approach illustrated in Figure 1. The reason for this is that in the sorting-based approach, one tends to merge very different states (e.g. the states $(6, 0)$ and $(0, 6)$, resulting in the merged state $(0, 0)$) which has a highly detrimental effect on the overall bound, while such a situation is avoided in the clustering approach to node merging. □

Our clustering-based node selection approach can be embedded in the top-down compilation for DDs in different ways. In the most natural variant, we assume that the number k of clusters to be created is equal to the maximum width W of the approximate DD. An alternative approach is to choose

$k < W$: Intuitively, this way the algorithm is able to explore more diverse parts of the solution space. Moreover, choosing k to be much smaller than W will lower the total running time because the clustering will not be used in every layer and because the final DD will be smaller.

Computational Results

In this section we present the results from computational experiments with our approach (i.e. clustering-based node selection) and compare it to that of minLP/maxSP approach (i.e. sortObj) for both variants discussed above, that is, $k = W$ and $k < W$, on two knapsack problems and three machine scheduling problems. We also experimented with alternative generic node-selection heuristics, e.g. random selection, maxState (sort nodes in KP based on state value) and minState (sort nodes in the scheduling problems based on minimum value among the time-related state representation coordinates). The obtained bounds, however, were worse than sortObj, which is itself dominated by our proposed approach, and thus are not reported below.

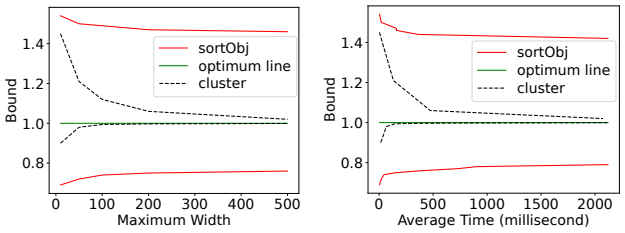
We implemented the approach in the Julia programming language (code is available here https://github.com/mnafar/aaai2024_clustering_DD), and we ran all the experiments on a Windows machine with processor 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz, 2.30 GHz and 16GB RAM. For the clustering part of the algorithm we use the implementation of the k-means algorithm from the *Clustering* package in Julia, setting the number of iterations to 50. We also performed supplementary experiments (results not reported below) with alternative clustering approaches such as k-medoids using Euclidean and squared Euclidean distance which yielded similar average results but showed a high variance in bound quality.

Note that for every problem presented next, the reported bounds and running times are the average bound and time taken over all of the considered instances.

Results for the First Variant: $k = W$

We start with the main variant in which the number k of clusters equals the maximum width of the approximate DD for five different discrete optimization problems. We proceed by briefly describing the problems including a sketch of the state information used for compiling the DD (and for the clustering, if not mentioned otherwise), the merge operator needed for relaxation, and the instances that were used. For each problem, we compare the primal and dual bounds obtained with our clustering approach to those obtained with the standard sortObj approach.

0/1 Knapsack Problem (KP). Given n items each having weight w_i and profit p_i, the goal is to select items that maximize the total profit such that the accumulated sum of the weights of the selected items does not exceed knapsack's capacity C. The state for compiling the DD is a positive integer representing the accumulated weight in a partial solution. A valid merge operator consists in choosing the state with minimum weight. The experiments are run on 100 KP instances with 200 items per instance taken from (Pisinger 2005). Figure 3 shows the average bounds obtained by sortObj versus those of obtained by clustering-based node selection using different maximum widths 3(a), and their running times in millisecond 3(b). In this figure, the green line displays the optimum, the red curves show the lower and upper bounds using sortObj and black dashed curves represent clustering-based node selection results. As becomes clear from the graphs of the experiments, our approach outperforms *sortObj* in all aspects, meaning it provides substantially better primal and dual bounds with smaller W (implying a smaller DD) and in a smaller computation time. Moreover, experimenting with instances involving 10000 items even with a maximum width of 1000, did not show scalability issues.

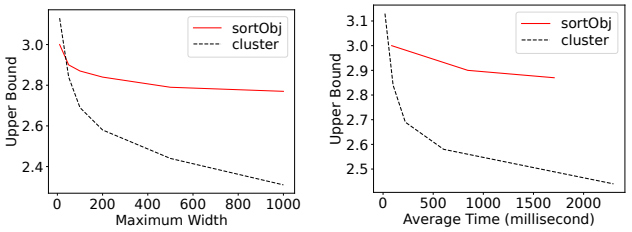


(a) bound vs W (b) bound vs Time

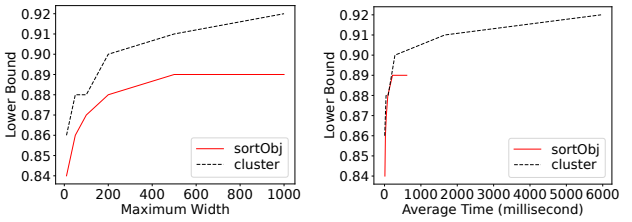
Figure 3: Clustering vs sortObj (KP)

Multidimensional Knapsack Problem (MKP). MKP is a generalization of the KP with multiple capacity constraints. An instance of MKP with n items and m dimensions has a capacity bound $(C_1, \cdots, C_m)$ for every dimension in which p_i and $(w_i^1, \cdots, w_i^m)$ are profits and weights for an item i. Similar to KP, the goal is to select a subset of items whose sum of profits is maximized such that all the capacity bound constraints hold simultaneously. In a DP model used for compiling a DD for the MKP, a state is an m-tuple in which every coordinate is the sum of the corresponding coordinates of the weights of the items that have been selected in the partial solution associated with the node under consideration. A valid merge operator for MKP is to take the element-wise minimum of the coordinates of the states to be merged. In case of the MKP, we did not only use the state coordinates as features for the clustering, but also the objective function value associated with each node in the DD. All experiments are run on 10 MKP instances where each of them is a 5-dimensional MKP with 100 items (taken from (Chu and Beasley 1998), which are publicly available in OR-LIBRARY at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html). Figure 4 shows the average bounds obtained by sortObj versus those obtained using clustering-based node selection for different maximum widths and their running times in milliseconds. Once again, it turns out that the clustering-based approach yields much better bounds than the standard sortObj approach both in relation to DD size and to the bound quality obtained in a certain amount of time.

Sum of Cubed Job Completion Times on Two Identical Machines $P_2||\Sigma C_j^3$. In an instance of $P_2||\Sigma C_j^3$, we are given n jobs, a job i has processing time p_i, and the goal

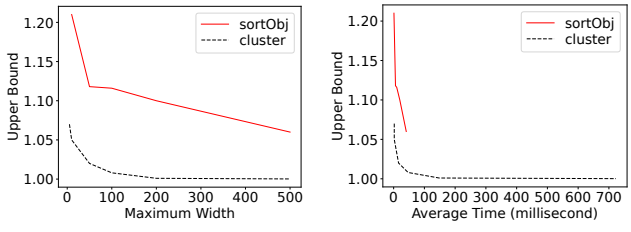


(a) upper bound vs W (b) upper bound vs time

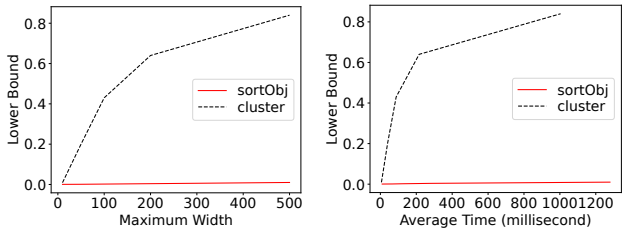


(c) lower bound vs W (d) lower bound vs time

Figure 4: Clustering vs sortObj (MKP)



(a) upper bound vs W (b) upper bound vs time



(c) lower bound vs W (d) lower bound vs time

Figure 5: Clustering vs sortObj ($P_2||\Sigma C_j^3$)

is to schedule these jobs on two identical machines such that the sum of the cubes of the job completion times is minimized. Every state in a DD for $P_2||\Sigma C_j^3$ is a 2-tuple where each of its coordinates represents the partial completion time on the corresponding machine. The merge operator is similar to that of MKP where a merged state is comprised of element-wise minimum of coordinates over the states that are being merged. The instances we ran the experiments on are from the set *wt100* that is publicly available from the OR-LIBRARY (http://people.brunel.ac.uk/~mastjjb/jeb/info.html). They were originally generated for weighted tardiness jobs on a single machine problem. We took the processing time of the jobs from the instances, and thus, the experiments are run on 125 instances each of which contains 100 jobs. The results of using sortObj and clustering-based node selection for building the corresponding DDs are shown in Figure 5. For this problem again the difference between performance of the two approaches is significant, i.e. the approach that uses clustering-based node selection is by far superior to the other approach.

Total Weighted Job Completion Time on Two Identical Machines $P_2||\Sigma w_j C_j$**.** A description of this problem is given in the example in Section . Once again, we performed experiments with the 125 instances from the OR Library used in $P_2||\Sigma C_j^3$. Figure 6 shows the performance of the two approaches for different maximum widths and their running times. We see that for this problem too our approach once again outperforms the baseline approach sortObj.

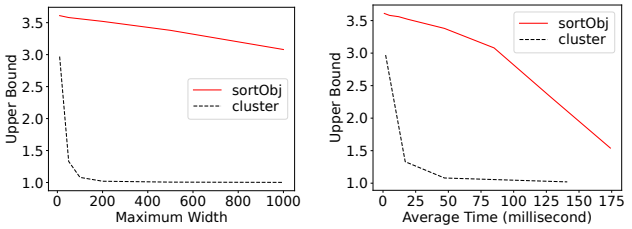
Weighted Number of Tardy Jobs on a Single Machine $1||\Sigma w_j U_j$**.** Given n jobs where p_i, w_i, and d_i are processing time, weight, and due date of a job i, the goal is to schedule the jobs on a single machine such that the weight of the tardy jobs is minimized. The state representation consists in a positive integer which measures the total processing time of the scheduled early jobs. Moreover, a merge operator for $1||\Sigma w_j U_j$ consists in choosing the state with minimum processing time. We experimented with 25 large instances with 500 jobs from (Tanaka, Fujikuma, and Araki 2009). Average bound and running time of different approaches ran on these instances are shown in Table 1, and they confirm the superiority of our approach which yields much better bounds with much smaller DD widths compared to the sortObj approach. Moreover, we performed experiments to obtain alternative bounds. We formulated the problem as MILP with positional variables (Keha, Khowala, and Fowler 2009). In our computational experiments, after an imposed time limit of 2 minutes, Gurobi found feasible solutions for all tested instances, but the both the primal and the dual bounds were far inferior than those obtained with our approach in less than one and two seconds (see Table 1).

clustering-based				sortObj				IP (Gurobi)	
W=**100**		**W=500**		W=**100**		**W=1000**			
time	primal	time	primal	time	primal	time	primal	time	primal
0.1	**1.36**	1.3	**1.2**	0.1	2.00	1.1	1.45	120	2.76
time	dual	time	dual	time	dual	time	dual	time	dual
0.1	**0.14**	1.6	**0.30**	2	0.05	2.7	0.23	120	0.06

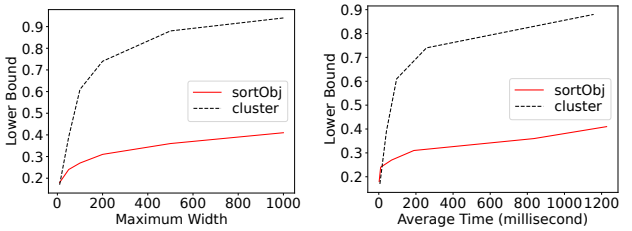
Table 1: Clustering, sortObj, and IP for $1||\Sigma w_j U_j$

Results for the Second Variant: $k < W$

Next, we will present the variant in which the number of clusters k is smaller than maximum width W of the approximate DD to be compiled. We illustrate the results for 0/1 Knapsack problem, noting that the results for the other problems follow similar patterns. In the following figures, the solid red curves represent DDs built using the sortObj node selection heuristic, and the dashed colorful curves correspond to DDs compiled using clustering-based node selection; every dashed curve corresponds to a specific maxi-



(a) upper bound vs W (b) upper bound vs time

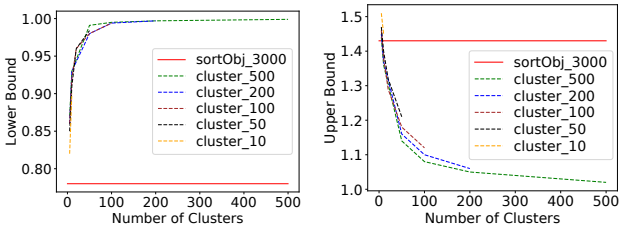


(c) lower bound vs W (d) lower bound vs time

Figure 6: Clustering vs sortObj ($P_2||\Sigma w_j C_j$)

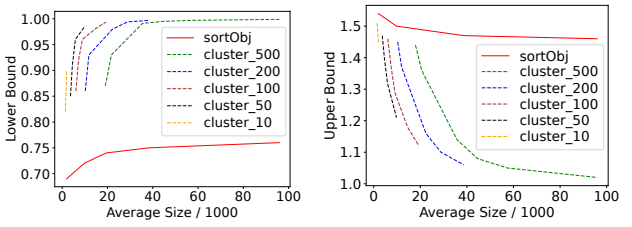
mum width W and different number k of clusters (e.g. $W = 200, k \in \{10, 50, 100, 200\}$ or $W = 50, k \in \{10, 20, 50\}$).

Figure 7 shows the bounds obtained from using different W and k for the KP. In the sortObj case, the maximum width is set to $W = 3000$, and for the clustering approach different maximum widths ($W \in \{10, 50, 100, 200, 500\}$) and different numbers k of clusters are taken.



(a) lower bound vs k (b) upper bound vs k

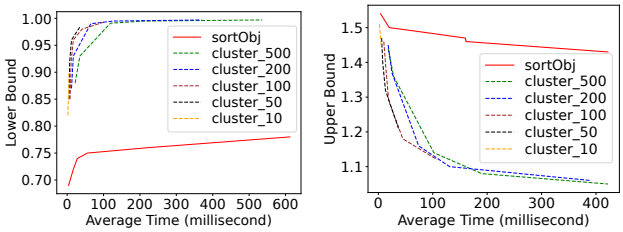
Figure 7: Clustering (dashed curves for various k) vs sortObj (red curves for $W = 3000$) for KP.



(a) lower bounds vs size (b) upper bounds vs size

Figure 8: Comparison of the sizes of the DDs

Figure 8 compares the sizes and the bounds for DDs compiled using clustering-based node selection and sortObj for KP. The range of maximum widths considered in the experiments is $[10, 500]$.



(a) lower bounds vs time (b) upper bounds vs time

Figure 9: Comparison of the times of the DDs

Figure 9 shows how changing the parameters for sortObj and clustering-based node selection affects their running times and compares these running times and the bounds for the two approaches applied on the KP. The ranges of the maximum widths are $[10, 1000]$ and $[10, 500]$ for sortObj and clustering-based node selection, respectively. The reason for having a larger W for sortObj is that we wanted to have the maximum running times be roughly equal for the sake of the readability of the graphs.

As it is clear in Figures 7, 8, and 9, making the choice to set $k < W$ does not decrease the quality of the bounds obtainable via DDs compiled using clustering-based node selection in a noticeable amount. However, it indeed decreases the required size and running time for providing bounds that are close to those achievable in the first variant. Therefore, this variant outperforms the sortObj baseline even more than the first variant in which $k = W$.

Conclusion

In this paper, we propose a novel and generic ML-based approach for node selection in the top-down compilation of approximate DDs that relies on clustering nodes according to their state information. We evaluated two variants of this approach on five different problem types, showing that it is able to provide substantially stronger bounds in relation to the size of the DD and the time needed to obtain the bounds than a similarly generic sorting-based approach that is commonly used in the literature.

It is important to note that all the results presented in this paper were obtained with a standard k-means clustering approach. In general, if one aims at improving the performance for certain problems, a natural approach would be to experiment with alternative clustering approaches, and to tune the parameters of the selected clustering approach.

Another natural extension of this research is to evaluate our approach within an exact DD-based solution approach, e.g. DD-based branch-and-bound. This would allow to see if the bound improvements achieved in this paper translate to a speed-up for exactly solving discrete optimization problems with DDs.

Acknowledgments

This research was funded by the Return Programme of the Federal State of North Rhine Westphalia (NRW Rückkehrprogramm).

References

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

Bergman, D.; Cire, A. A.; Van Hoeve, W.-J.; and Hooker, J. N. 2016. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1): 47–66.

Bruno, J.; Coffman Jr, E. G.; and Sethi, R. 1974. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7): 382–387.

Cappart, Q.; Bergman, D.; Rousseau, L.-M.; Prémont-Schwarz, I.; and Parjadis, A. 2022. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 34(5): 2552–2570.

Cappart, Q.; Chételat, D.; Khalil, E. B.; Lodi, A.; Morris, C.; and Velickovic, P. 2023. Combinatorial optimization and reasoning with graph neural networks. *J. Mach. Learn. Res.*, 24: 130–1.

Cappart, Q.; Goutierre, E.; Bergman, D.; and Rousseau, L.-M. 2019. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 1443–1451.

Castro, M. P.; Cire, A. A.; and Beck, J. C. 2022. Decision diagrams for discrete optimization: A survey of recent advances. *INFORMS Journal on Computing*, 34(4): 2271–2295.

Chu, P. C.; and Beasley, J. E. 1998. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4: 63–86.

Coppé, V.; Gillard, X.; and Schaus, P. 2023. Boosting Decision Diagram-Based Branch-And-Bound by Pre-Solving with Aggregate Dynamic Programming. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

de Weerdt, M.; Baart, R.; and He, L. 2021. Single-machine scheduling with release times, deadlines, setup times, and rejection. *European Journal of Operational Research*, 291(2): 629–639.

Frohner, N.; and Raidl, G. R. 2019. Merging quality estimation for binary decision diagrams with binary classifiers. In *International Conference on Machine Learning, Optimization, and Data Science*, 445–457. Springer.

Graham, R.; Lawler, E.; Lenstra, J.; and Kan, A. 1979. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In Hammer, P.; Johnson, E.; and Korte, B., eds., *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, 287–326. Elsevier.

Hooker, J. N. 2013. Decision diagrams and dynamic programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10*, 94–110. Springer.

Hooker, J. N. 2017. Job sequencing bounds from decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, 565–578. Springer.

Horn, M.; Maschler, J.; Raidl, G. R.; and Rönnberg, E. 2021. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research*, 126: 105125.

Hottung, A.; Kwon, Y.-D.; and Tierney, K. 2021. Efficient active search for combinatorial optimization problems. *arXiv preprint arXiv:2106.05126*.

Keha, A. B.; Khowala, K.; and Fowler, J. W. 2009. Mixed integer programming formulations for single machine scheduling problems. *Computers & Industrial Engineering*, 56(1): 357–367.

Kellerer, H.; Pferschy, U.; and Pisinger, D. 2004. *Knapsack Problems*. Springer, Berlin, Germany.

Khalil, E.; Le Bodic, P.; Song, L.; Nemhauser, G.; and Dilkina, B. 2016. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Kotary, J.; Fioretto, F.; Van Hentenryck, P.; and Wilder, B. 2021. End-to-end constrained optimization learning: A survey. *arXiv preprint arXiv:2103.16378*.

Lenstra, J. K.; Kan, A. R.; and Brucker, P. 1977. Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, 343–362. Elsevier.

Parjadis, A.; Cappart, Q.; Rousseau, L.-M.; and Bergman, D. 2021. Improving branch-and-bound using decision diagrams and reinforcement learning. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, 446–455. Springer.

Pisinger, D. 2005. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9): 2271–2284.

Tanaka, S.; Fujikuma, S.; and Araki, M. 2009. An exact algorithm for single-machine scheduling without machine idle time. *Journal of Scheduling*, 12: 575–593.