Boosting Decision Diagram-Based Branch-And-Bound by Pre-Solving with Aggregate Dynamic Programming

Vianney Coppé
UCLouvain, Louvain-la-Neuve, Belgium

Xavier Gillard
UCLouvain, Louvain-la-Neuve, Belgium

Pierre Schaus
UCLouvain, Louvain-la-Neuve, Belgium

Abstract

Discrete optimization problems expressible as dynamic programs can be solved by branch-and-bound with decision diagrams. This approach dynamically compiles bounded-width decision diagrams to derive both lower and upper bounds on unexplored parts of the search space, until they are all enumerated or discarded. Assuming a minimization problem, relaxed decision diagrams provide lower bounds through state merging while restricted decision diagrams obtain upper bounds by excluding states to limit their size. As the selection of states to merge or delete is done locally, it is very myopic to the global problem structure. In this paper, we propose a novel way to proceed that is based on pre-solving a so-called aggregate version of the problem with a limited number of states. The compiled decision diagram of this aggregate problem is tractable and can fit in memory. It can then be exploited by the original branch-and-bound to generate additional pruning and guide the compilation of restricted decision diagrams toward good solutions. The results of the numerical study we conducted on three combinatorial optimization problems show a clear improvement in the performance of DD-based solvers when blended with the proposed techniques. These results also suggest an approach where the aggregate dynamic programming model could be used in replacement of the relaxed decision diagrams altogether.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Discrete Optimization, Decision Diagrams, Aggregate Dynamic Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2023.13

1 Introduction

On top of their use for Boolean encodings [27], formal verification [25], model checking [15], computer-aided design [29] and much more, *decision diagrams* (DDs) have recently emerged as a tool for discrete optimization. They provide a compact way to encode a set of solutions to a problem. Still, for large problems, DDs representing the whole solution space – called *exact* DDs – can quickly become intractable to compute. Two variants of DDs can be used instead: *restricted* [10] and *relaxed* [1, 8] DDs that respectively encode a subset and superset of the set of solutions. When compiled based on a *dynamic programming* (DP) model, these approximate DDs allow to compute bounds on the objective function for any subproblem while controlling the size of the DD compiled. Restricted DDs aim to find good admissible solutions by iteratively extending a bounded set of promising candidates while dropping others, in a beam search fashion. On the other hand, relaxed DDs rely on a problem-dependent state merging scheme to maintain an acceptable DD size while preserving all solutions of the problem. In [9], Bergman et al. presented a *branch-and-bound* algorithm solely based on these two ingredients, thus introducing a new general-purpose discrete optimization framework and solver.

In addition to exploiting the compactness of DP models, the main novelty of this approach is its unique way of deriving lower and upper bounds. In the last few years, some algorithmic improvements have been suggested to further strengthen these bounds. Assuming a minimization problem, Gillard et al. [19] showed how user-defined lower bound formulas can be integrated to prune DDs during their compilation and thus concentrate the search on promising parts of the search space. They also proposed a way to compute tighter lower bounds for all nodes contained in a relaxed DD through *local bounds*. Rudich et al. [30] introduced a *peeling* operator that splits a relaxed DD in two: one part containing all paths traversing a selected exact node and the other containing all remaining paths. It allows both to warm-start the compilation of subsequent relaxed DDs and to strengthen the bounds of the nodes inside the relaxed DD on which the peeling has been performed. More recently, [16] generalized the ideas of [19] and introduced the use of a cache storing new thresholds that further enhance the pruning power of the solver. Other factors impacting the quality of the bounds provided by relaxed DDs have been studied, including variable orderings [7, 11, 26] and alternative compilation schemes [24]. Yet, all these approaches rely on a problem-specific state merging operator at the heart of the relaxation, which does not yield tight relaxations for all problems, as our computational experiments show.

After covering the necessary background about DD-based optimization, this paper presents an alternate relaxation scheme for deriving good bounds by incorporating ideas from *aggregate dynamic programming* [2, 3] to the DD-based discrete optimization framework. The underlying idea of the approach is to deduce information about an original problem instance by creating and solving an aggregate – relaxed – version of it. This is achieved by *aggregating* the states of the DP model as to obtain a much smaller DP state space. If this aggregation is adequately specified, one can compute a lower bound for any original subproblem by finding the optimal solution of its aggregate version. Furthermore, this optimal aggregate solution can be *disaggregated* and transposed in the original problem to find good heuristic solutions. In practice, the aggregation-based lower bounds are used as additional pruning within the compilation of relaxed and restricted DDs. Moreover, aggregate solutions are translated into node selection heuristics to steer the compilation of restricted DDs toward resembling solutions to the original problem, which are thus expected to be good.

Throughout the paper, the framework is illustrated on three different combinatorial problems: the Talent Scheduling Problem, the Pigment Sequencing Problem and the Aircraft Landing Problem. They are then used for the experimental evaluation of the framework, the results of which show that the aggregation-based bound brings additional pruning and enables solving more instances. Furthermore, the aggregation-based node selection heuristic improves the quality of the solutions found early in the search and thus contributes to speeding up the overall resolution. Finally, we show that a DD-based solver using only the aggregation-based bound as relaxation performs almost equally well, which is a promising direction for problems for which defining a merging operator is difficult or inefficient.

Although this paper is – to the best of our knowledge – the first to combine aggregate dynamic programming with the DD-based branch-and-bound paradigm proposed by Bergman et al, there has already been some hybridization work to combine discrete optimization with DDs and other methods. For instance, in [12], Cappart et al. propose to use reinforcement learning to guess the variable ordering that should be used to derive the best possible bounds from the compiled approximate DDs. Other attempts combined DDs with Lagrangian relaxation [13, 23] or MIP [5, 22, 31, 32]. On a slightly different note, a method has been proposed where restricted DDs are used to generate good neighborhoods in a *large neighborhood search* framework [20].

2 Preliminaries

2.1 Discrete Optimization

A discrete optimization problem $\mathcal{P}$ involves finding the best possible solution x^* from a finite set of feasible solutions $Sol(\mathcal{P}) = D \cap C$. This set is determined by the domain $D = D_0 \times \dots \times D_{n-1}$ from which the variables $x = \langle x_0, \dots, x_{n-1} \rangle$ each take on a value, i.e. $x_j \in D_j$, and by a set of constraints C imposed on the value assignments. The quality of the solutions is evaluated according to an objective function $f(x)$ that must be minimized. Formally, the problem is defined as $\min \{f(x) \mid x \in D \cap C\}$ and any optimal solution x^* must satisfy $x^* \in Sol(\mathcal{P})$ and $\forall x \in Sol(\mathcal{P}) : f(x^*) \leq f(x)$. We describe below three optimization problems that will be utilized in the paper as illustrations for the aggregation-based framework.

Talent Scheduling Problem. The *Talent Scheduling Problem* (TalentSched) is a film shoot scheduling problem that considers a set $N = \{0, \dots, n-1\}$ of scenes and a set $A = \{0, \dots, m-1\}$ of actors. Each scene $i \in N$ involves a required set $R_i \subseteq A$ of actors for a duration $D_i \in \mathbb{N}$. Moreover, each actor $k \in M$ has a pay rate C_k and is paid without interruption from their first to their last scheduled scene. The objective of TalentSched is to find a permutation of the scenes that minimizes the total cost of the film shoot.

Pigment Sequencing Problem. The *Pigment Sequencing Problem* (PSP) is a single-machine production planning problem that aims to minimize the stocking and changeover costs while satisfying a set of orders. There are different item types $I = \{0, \dots, n-1\}$ with a given stocking cost S_i to pay for each time period between the production and the deadline of an order. For each pair $i, j \in I$ of item types, a changeover cost C_{ij} is incurred whenever the machine switches the production from item type i to j. Finally, the demand matrix Q contains all the orders: $Q_p^i \in \{0, 1\}$ indicates whether there is an order for item type $i \in I$ at time period p with $0 \leq p < H$ and H the time horizon.

Aircraft Landing Problem. The *Aircraft Landing Problem* (ALP) requires to schedule the landing of a set of aircrafts $N = \{0, \dots, n-1\}$ on a set of runways $R = \{0, \dots, r-1\}$. The aircrafts have a target T_i and latest L_i landing time. Moreover, the set of aircrafts is partitioned in disjoint sets $A_0, \dots, A_{c-1}$ corresponding to different aircraft classes in $C = \{0, \dots, c-1\}$. For each pair of aircraft classes $a, b \in C$, a minimum separation time $S_{a,b}$ between the landings is given. The goal is to find the schedule that minimizes the total waiting time of the aircrafts – the delay between their target time and scheduled landing time – while respecting their latest landing time.

2.2 Dynamic Programming

Dynamic programming (DP) is a *divide-and-conquer* strategy introduced by Bellman [4] for solving discrete optimization problems with an inherent recursive structure. It works by recursively decomposing the problem in smaller and overlapping subproblems. The cornerstone of the approach is the caching of intermediate results that allows each distinct subproblem to be solved only once. A *DP model* of a discrete optimization problem $\mathcal{P}$ can be defined as a labeled transition system consisting of:

- the *control variables* $x_j \in D_j$ with $j \in \{0, \dots, n-1\}$.
- a set of *state-spaces* $S = \{S_0, \dots, S_n\}$ among which one distinguishes the *initial state* r, the *terminal state* t and the *infeasible state* $\hat{0}$.

- a set t of *transition functions* s.t. $t_j : S_j \times D_j \to S_{j+1}$ for $j = 0, \dots, n-1$ taking the system from one state s^j to the next state s^{j+1} based on the value d assigned to variable x_j, or to $\perp$ if assigning $x_j = d$ is infeasible. These functions should never allow one to recover from infeasibility, i.e. $t_j(\hat{0}, d) = \hat{0}$ for any $d \in D_j$.
- a set h of *transition value functions* s.t. $h_j : S_j \times D_j \to \mathbb{R}$ representing the immediate reward of assigning some value $d \in D_j$ to the variable x_j for $j = 0, \dots, n-1$.
- a *root value* v_r.

On that basis, the objective function $f(x)$ of $\mathcal{P}$ is formulated as follows:

$$\begin{aligned} &\text{minimize } f(x) = v_r + \sum_{j=0}^{n-1} h_j(s^j, x_j) \\ &\text{subject to } s^{j+1} = t_j(s^j, x_j), \text{ for all } j = 0, \dots, n-1, \text{ with } x_j \in D_j \\ &\qquad\qquad s^j \in S_j, j = 0, \dots, n \text{ and } x \in C. \end{aligned} \tag{1}$$

TalentSched. A DP model for TalentSched was introduced in [17] that we slightly adapt here to make it suitable for the relaxation discussed in Section 2.3.1. States of this model are pairs (M, P) where M and P are disjoint sets of scenes that respectively must or might still be scheduled. The only case where P is non-empty happens when a state is relaxed.

- Control variables: $x_j \in N$ with $0 \leq j < n$ decides which scene is shot in j-th position.
- State spaces: $S = \{(M, P) \mid M, P \subseteq N, M \cap P = \emptyset\}$. The root state is $r = (N, \emptyset)$ and the terminal states are of the form $(\emptyset, P)$.
- Transition functions:

$$t_j(s^j, x_j) = \begin{cases} (s^j.M \setminus \{x_j\}, s^j.P \setminus \{x_j\}) & \text{if } x_j \in s^j.M, \\ (s^j.M \setminus \{x_j\}, s^j.P \setminus \{x_j\}), & \text{if } x_j \in s^j.P \text{ and } |s^j.M| < n - j, \\ \hat{0}, & \text{otherwise.} \end{cases}$$

  A scene from P can only be selected if there are more spots left than scenes in M.
- Transition value functions: let $a(Q) = \cup_{i \in Q} R_i$ be the required set of actors for a set of scenes Q. Given a state $s = (M, P)$, the set of actors that are guaranteed to be on-location is computed as $o(s) = a(s.M) \cap a(N \setminus (s.M \cup s.P))$ because they are required both for a scene that must still be scheduled and for another that is guaranteed to be scheduled. In the transition value functions, we add all the actors from R_{x_j} to this set and sum the individual costs: $h_j(s^j, x_j) = D_{x_j} \sum_{k \in o(s^j) \cup R_{x_j}} C_k$.
- Root value: $v_r = 0$.

PSP. The PSP was already tackled with a DD-based approach in [16, 20]. We hereby recall the DP model from [16] that allows the machine to be idle at some time periods. In this model, the decisions are made backwards – this allows to define transition functions that only lead to feasible production schedules. If variable x_j decides the type of item to produce at period j, the reverse variable ordering $x_{H-1}, \dots, x_0$ is thus used. To simplify the transition functions, let us denote by P^i_r the time period at which the r-th item of type i must be delivered, i.e. $P^i_r = \min\{0 \leq q < H \mid \sum_{p=0}^{q} Q^i_p \geq r\}$ for all $i \in N, 0 \leq r \leq \sum_{0 \leq p < H} Q^i_p$. Moreover, we define a dummy item type $\perp$ used for idle periods and $N' = N \cup \{\perp\}$.

States are pairs (i, R) with i the item type that the machine is currently set to produce and R a vector that gives the remaining number R_i of demands to satisfy for each type i.

- Control variables: $x_j \in N'$ with $0 \leq j < H$ decides the item type to produce at period j.

- State space: $S = \{s \mid s.i \in N', \forall i \in N, 0 \leq s.R_i \leq \sum_{0 \leq p < H} Q_p^i\}$. The root state is given by $r = \langle \perp, (\sum_{0 \leq p < H} Q_p^0, \ldots, \sum_{0 \leq p < H} Q_p^{n-1})\rangle$ and the terminal states are of the form $\langle i, (0, \ldots, 0)\rangle$ with $i \in N'$.
- Transition functions:

$$t_j(s^j, x_j) = \begin{cases} \left\langle t_j^i(s^j, x_j), t_j^R(s^j, x_j)\right\rangle, & \text{if } x_j \neq \perp \text{ and } s^j.R_{x_j} > 0 \text{ and } j \leq P^{x_j}_{s^j.R_{x_j}}, \\ \left\langle t_j^i(s^j, x_j), t_j^R(s^j, x_j)\right\rangle, & \text{if } x_j = \perp \text{ and } \sum_{i \in N} s^j.R_i < j + 1, \\ \hat{0}, & \text{otherwise.} \end{cases}$$

where

$$\begin{aligned} t_j^i(s^j, x_j) &= \begin{cases} x_j, & \text{if } x_j \neq \perp \\ s^j.i, & \text{otherwise.} \end{cases} \\ t_j^R(s^j, x_j) &= \begin{cases} (s^j.R_0, \ldots, s^j.R_{x_j} - 1, \ldots, s^j.R_{n-1}), & \text{if } x_j \neq \perp \\ s^j.R, & \text{otherwise.} \end{cases} \end{aligned}$$

In the transition function, the first condition ensures that there remains at least one item to produce for the chosen type and that the current time period j is earlier than its deadline. The second condition ensures that idle periods cannot be scheduled when the remaining quantity to produce is equal to the number of periods left.
- Transition value functions: the changeover and stocking costs are computed as:

$$h_j(s^j, x_j) = \left\{ \begin{array}{ll} C_{x_j s^j.i}, & \text{if } x_j \neq \perp \text{ and } s^j.i \neq \perp \\ 0, & \text{otherwise.} \end{array} \right\} + \left\{ \begin{array}{ll} S_{x_j} \cdot (j - P^{x_j}_{s^j.R_{x_j}}), & \text{if } x_j \neq \perp \\ 0, & \text{otherwise.} \end{array} \right\}$$

- Root value: $v_r = 0$.

ALP. We reproduce here the DP model presented in [28] where states are pairs (Q, ROP), with Q a vector that gives the remaining number of aircrafts of each class to schedule and ROP a *runway occupation profile*: a vector containing pairs (l, c) that respectively give the time and aircraft class of the latest landing scheduled on each runway. Similarly to the PSP modeling, we denote by $\perp$ either a dummy aircraft class or a dummy runway.

- Control variables: we use pairs of variables $(x_j, y_j) \in (C \times R) \cup \{(\perp, \perp)\}$ with $0 \leq j < n$ that represent the decision to place an aircraft of class x_j on runway y_j, or to schedule nothing at all in case of $(\perp, \perp)$.
- State spaces:
$S = \{(Q, ROP) \mid \forall i \in C : Q_i \geq 0, \forall k \in R : ROP_k.l \geq 0, ROP_k.c \in C \cup \{\perp\}\}$. The root state is $r = (\langle |A_0|, \ldots, |A_{c-1}|\rangle, \langle (0, \perp), \ldots, (0, \perp)\rangle)$ and the terminal states are of the form $(\langle 0, \ldots, 0\rangle, ROP)$.
- Transition functions: if A_i^k gives the aircraft from class i that must be scheduled when there are k aircrafts left from this class, we can define the function computing the earliest landing time given a state s, a class x and a runway y:

$$E(s, x, y) = \begin{cases} T_{A_x^{s.Q_x}}, & \text{if } s.ROP_y.l = 0 \text{ and } s.ROP_y.c = \perp, \\ \max(s.ROP_y.l + \min_{i \in C} S_{i,x}, T_{A_x^{s.Q_x}}), & \text{if } s.ROP_y.l > 0 \text{ and } s.ROP_y.c = \perp, \\ \max(s.ROP_y.l + S_{s.ROP_y.c,x}, T_{A_x^{s.Q_x}}), & \text{otherwise.} \end{cases}$$

This allows us to define the transition functions as:

$$t_j(s^j, x_j, y_j) = \begin{cases} (t_j^Q(s^j, x_j, y_j), t_j^{ROP}(s^j, x_j, y_j)), & \begin{array}{l} \text{if } x_j \neq \perp \ \text{ and } s^j.Q_{x_j} > 0 \\ \qquad\qquad \text{and } E(s^j, x_j, y_j) \leq L_{A_{x_j}^{s^j.Q_{x_j}}}, \end{array} \\ (t_j^Q(s^j, x_j, y_j), t_j^{ROP}(s^j, x_j, y_j)), & \text{if } x_j = \perp \text{ and } \sum_{i \in C} s^j.Q_i = 0, \\ \hat{0}, & \text{otherwise.} \end{cases}$$

where

$$t_j^Q(s^j, x_j, y_j) = \begin{cases} \langle s^j.Q_0, \ldots, s^j.Q_{x_j} - 1, \ldots, s^j.Q_{c-1} \rangle, & \text{if } x_j \neq \bot \\ s^j.Q, & \text{otherwise.} \end{cases}$$

$$t_j^{ROP}(s^j, x_j, y_j) = \begin{cases} \langle s^j.ROP_0, \ldots, (E(s^j, x_j, y_j), x_j), \ldots, s^j.ROP_{r-1} \rangle, & \text{if } x_j \neq \bot \\ s^j.ROP, & \text{otherwise.} \end{cases}$$

The first condition of the transition function ensures that there remains at least one aircraft of the chosen class and that its earliest landing time is not greater its latest landing time. The second condition only allows us to schedule dummy aircrafts when there are no aircrafts left to schedule.

- Transition value functions: the waiting time of the aircraft is computed as:

$$h_j(s^j, x_j, y_j) = \begin{cases} E(s^j, x_j, y_j) - T_{A_{x_j}}^{s^j.Q_{x_j}}, & \text{if } x_j \neq \bot \\ 0, & \text{otherwise.} \end{cases}$$

- Root value: $v_r = 0$.

Because the runways are identical and independent, there are many symmetries in this model. This can be mitigated by sorting the ROP of every state by increasing latest landing time, breaking ties according to the previous aircraft class scheduled.

2.3 Decision Diagrams

When used to manipulate the DP model of a discrete optimization problem $\mathcal{P}$, DDs are graphical encodings that represent a set of solutions of the problem. More precisely, a DD $\mathcal{B} = (U, A, \sigma, l, v)$ is a layered directed acyclic graph composed of a set of nodes U interconnected by a set of arcs A. Starting from a single node u_r corresponding to a DP state given by the function $\sigma(u_r)$, the process of iteratively extending a set of partial solutions is called the *compilation* of a DD and is described by Algorithm 1. Note that the highlighted portions concern the ingredients introduced in Section 3 and can be ignored for now. The algorithm begins by initializing a layer L_i that only contains the *root node* u_r, assuming its state $\sigma(u_r)$ belongs to the i-th stage of the DP model. The subsequent layers of the DD are then constructed sequentially by applying each valid transition of the DP model to every node of the last completed layer at lines 8–16. Each layer thus corresponds to a stage of the DP model and contains a single node for each state reached in order to preserve the compactness of the model. The arcs $a \in A$ materialize the transitions that exist between the states of consecutive stages. In particular, the arc $a = (u \xrightarrow{d} u')$ connecting nodes $u \in L_j, u' \in L_{j+1}$ represents the transition between $\sigma(u)$ and $\sigma(u')$. The decision associated with this transition is stored by the *label* $l(a) = d \in D_j$ and the transition value is given by the arc *value* $v(a)$.

The algorithm completes when the last layer L_n is generated, constituted by a single node t called the *terminal* node. The DD thus constructed contains a set of $u_r \rightsquigarrow t$ paths that can be combined with any previously discovered $r \rightsquigarrow u_r$ path, connecting the *root* of the problem to u_r. Any $r \rightsquigarrow t$ path $p = (a_0, \ldots, a_{n-1})$ represents a solution given by $x(p) = (l(a_0), \ldots, l(a_{n-1}))$. The objective value of such solution can also be retrieved from the sequence of arcs by accumulating their values, and adding the root value: $v(p) = v_r + \sum_{j=0}^{n-1} v(a_j)$. The set of solutions contained in the DD is denoted as $Sol(\mathcal{B}) = \{x(p) \mid \exists p : r \rightsquigarrow t, p \in \mathcal{B}\}$. A DD rooted at a node u_r is *exact* if it perfectly represents the set of solutions of the corresponding subproblem $\mathcal{P}|_{u_r}$, i.e. $Sol(\mathcal{B}) = Sol(\mathcal{P}|_{u_r})$ and $v(p) = f(x(p)), \forall p \in \mathcal{B}$. The best value among the $u_1 \rightsquigarrow u_2$ paths in $\mathcal{B}$ is denoted $v^*(u_1 \rightsquigarrow u_2 \mid \mathcal{B})$, and in particular $v^*(u \mid \mathcal{B}) = v^*(r \rightsquigarrow u \mid \mathcal{B})$.

Algorithm 1 Compilation of DD $\mathcal{B}$ rooted at node u_r with maximum width W.

```
1:  $i \leftarrow$ index of the layer containing $u_r$
2:  $L_i \leftarrow \{u_r\}$
3:  $\widetilde{P} \leftarrow \Delta(\widetilde{p})$ with $\widetilde{p}$ the optimal solution for $\pi(\sigma(u_r))$    // retrieve disaggregate solution
4:  for $j = i$ to $n-1$ do
5:    if $|L_j| > W$ then
6:      restrict or relax the layer to get $W$ nodes with Algorithm 2
7:    $L_{j+1} \leftarrow \emptyset$
8:    for all $u \in L_j$ do
9:      $\underline{v}_{rlb}(\sigma(u)) \leftarrow \max\left\{\underline{v}_{rlb}(\sigma(u)), \underline{v}_{agg}(\pi(\sigma(u)))\right\}$    // inject aggregation-based bound
10:     if $v^*(u \mid \mathcal{B}) + \underline{v}_{rlb}(\sigma(u)) \geq \overline{v}$ then // rough lower bound pruning w.r.t. incumbent
11:       continue
12:     for all $d \in D_j$ do
13:       create node $u'$ with state $\sigma(u') = t_j(\sigma(u), d)$ or retrieve it from $L_{j+1}$
14:       create arc $a = (u \xrightarrow{d} u')$ with $v(a) = h_j(\sigma(u), d)$ and $l(a) = d$
15:       $score(a) \leftarrow 1$ if $l(a) \in \widetilde{P}_j$, 0 otherwise
16:       add $u'$ to $L_{j+1}$ and add $a$ to $A$
```

▶ **Example 1.** Let us define a TalentSched instance with 4 scenes with durations $D = \langle 3, 5, 2, 4\rangle$ and 4 actors with pay rates $C = \langle 10, 20, 30, 40\rangle$. The actor requirements for each scene are given by $R = \langle\{0, 3\}, \{0, 1, 3\}, \{0, 2, 3\}, \{0, 1, 2, 3\}\rangle$. Figure 1 shows the exact DD compiled for this instance with the DP model recalled in Section 2.2. Note that for each state $s = (M, P)$ corresponding to a node in the DD, we only show the set M since P is always empty in exact nodes. An optimal solution of the problem is $\langle 0, 2, 3, 1\rangle$, which gives an objective value of 106.

As the reader might have guessed, the compilation of an exact DD for a combinatorial optimization problem suffers from the curse of dimensionality as much as the corresponding DP model. This is why DD-based discrete optimization rarely relies on exact DDs but rather on *restricted* and *relaxed* DDs. These two variants follow two distinct compilation schemes that allow to maintain the number of nodes of each layer – called the *width* – under a given parameter W. In Algorithm 1, this logic is performed at line 5 where the width of the current layer is compared with W. If needed, the layer is then either restricted or relaxed at line 6 by calling Algorithm 2.

2.3.1 Approximate Decision Diagrams

As stated by Algorithm 2, restricted DDs simply remove surplus nodes from the layer until it is reduced to W nodes. A heuristic is used to evaluate the nodes and drop the least promising ones. Restricted DDs thus generate a subset of the solutions of the corresponding problem, i.e. $Sol(\overline{\mathcal{B}}) \subseteq Sol(\mathcal{P})$ and $v(p) = f(x(p)), \forall p \in \overline{\mathcal{B}}$ for a restricted DD $\overline{\mathcal{B}}$. They thus provide upper bounds on the objective value.

As opposed to restricted DDs, a relaxed DD $\underline{\mathcal{B}}$ yields lower bounds by representing a superset of the solutions of the corresponding problem: $Sol(\underline{\mathcal{B}}) \supseteq Sol(\mathcal{P})$ and $v(p) \leq f(x(p)), \forall p \in \underline{\mathcal{B}}$. This is achieved through a problem-specific *state merging* operator $\oplus(\sigma(\mathcal{M}))$ that defines an approximate representation that includes all states $\sigma(\mathcal{M}) = \{\sigma(u) \mid u \in \mathcal{M}\}$ corresponding to the merged nodes $\mathcal{M}$ and preserves all their outgoing transitions, although

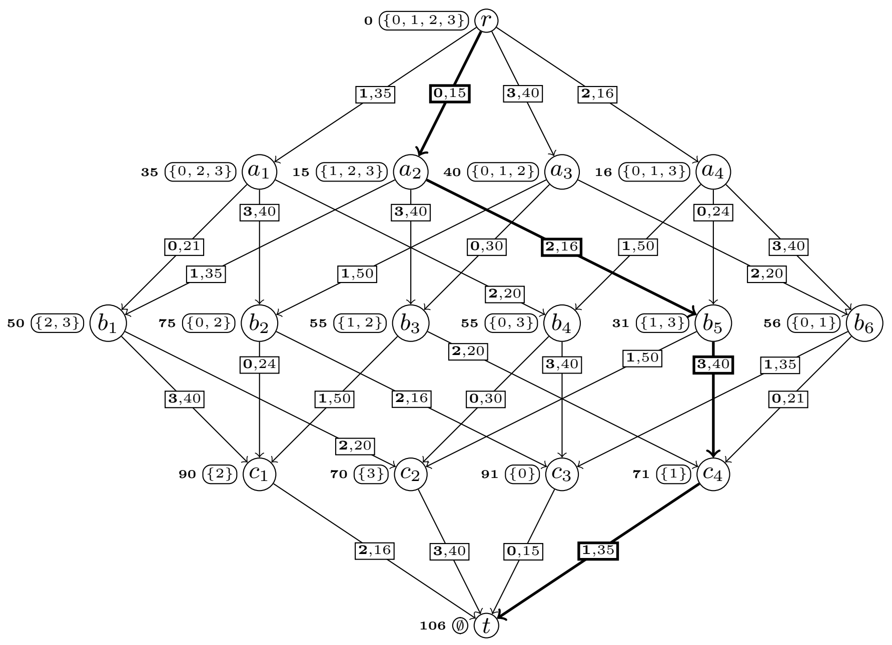

Figure 1 The exact DD for the TalentSched instance given in Example 1. Nodes are annotated with their state and the best prefix value. Arcs are labeled with the associated decision in bold and transition value. The arcs constituting one of the optimal solutions are highlighted in bold.

Algorithm 2 Restriction or relaxation of layer L_j with maximum width W.

```
1: while $|L_j| > W$ do
2:     $\mathcal{M} \leftarrow$ select nodes from $L_j$ according to their *score*
3:     $L_j \leftarrow L_j \setminus \mathcal{M}$
4:     create node $\mu$ with state $\sigma(\mu) = \oplus(\sigma(\mathcal{M}))$ and add it to $L_j$    // for relaxation only
5:     for all $u \in \mathcal{M}$ and arc $a = (u' \xrightarrow{d} u)$ incident to $u$ do
6:         replace $a$ by $a' = (u' \xrightarrow{d} \mu)$ and set $v(a') = \Gamma_{\mathcal{M}}(v(a), u)$
```

it may also introduce infeasible transitions. In Algorithm 2, a *meta*-node is created for this merged state at line 4 and the arcs pointing to the deleted nodes are redirected to this merged node at line 6. The operator $\Gamma_{\mathcal{M}}$ permits to adjust the value of these arcs if needed. In all three formulations given below, this operator is the identity function.

TalentSched. The merging operator is defined by $\oplus(\mathcal{M}) = (\oplus_M(\mathcal{M}), \oplus_P(\mathcal{M}))$ where $\oplus_M(\mathcal{M}) = \bigcap_{s \in \mathcal{M}} s.M$ and $\oplus_P(\mathcal{M}) = (\bigcup_{s \in \mathcal{M}} s.M \cup s.P) \setminus (\bigcap_{s \in \mathcal{M}} s.M)$. The definition of $\oplus_P(\mathcal{M})$ ensures that the resulting set of scenes that might be scheduled contains any scene that must or might be scheduled in any of the states, except those that still must be scheduled for all states.

PSP. A valid merging operator is $\oplus(\mathcal{M}) = (\bot, \langle \min_{s \in \mathcal{M}} s.R_0, \ldots, \min_{s \in \mathcal{M}} s.R_{n-1} \rangle)$. The configuration of the machine is always reset to the dummy item type $\bot$ as there is little chance that merged states agree on it. For each item type, the remaining number of demands is computed by taking the minimum value among all merged states, meaning that any demand satisfied by at least one state is considered satisfied in the merged state.

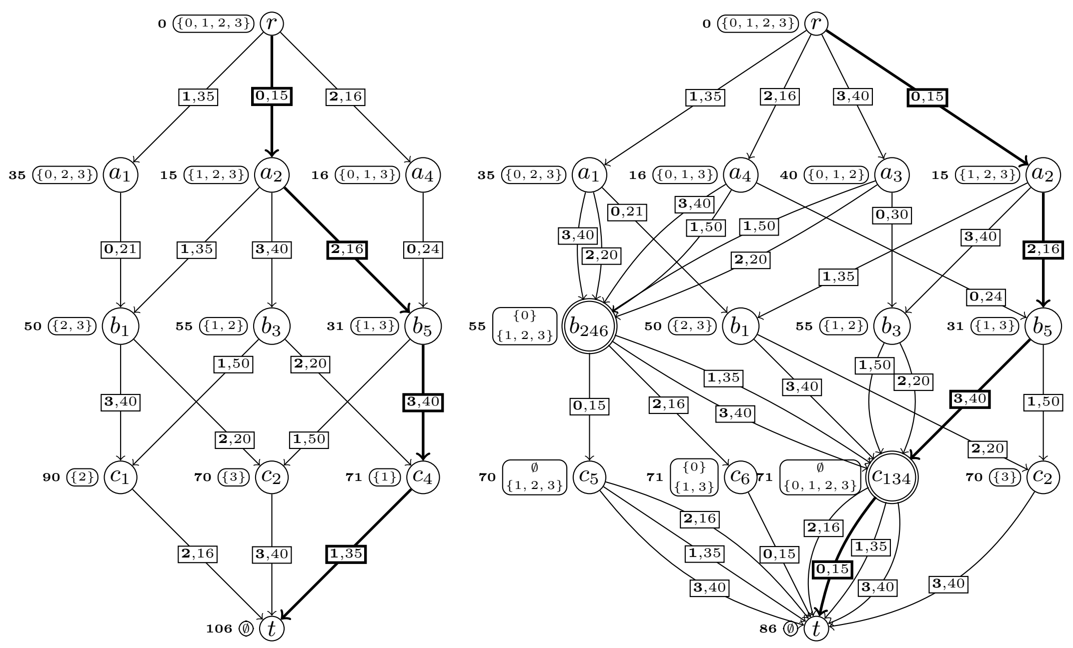

Figure 2 Respectively on the left and the right, a restricted and relaxed DD for the TalentSched instance given in Example 1, compiled with W set to 3 and 4. Merged nodes are circled twice.

ALP. The merging operator is again defined separately for each component of the states: $\oplus(\mathcal{M}) = (\oplus_Q(\mathcal{M}), \oplus_{ROP}(\mathcal{M}))$. First, the minimum remaining quantity of aircrafts for each class is stored in the merged state: $\oplus_Q(\mathcal{M}) = \langle \min_{s\in\mathcal{M}} s.Q_0, \ldots, \min_{s\in\mathcal{M}} s.Q_{c-1}\rangle$. For the ROP, the minimum latest landing time on each runway is kept and the last aircraft classes scheduled are reset to $\perp$: $\oplus_{ROP}(\mathcal{M}) = \langle(\min_{s\in\mathcal{M}} s.ROP_0.l, \perp), \ldots, (\min_{s\in\mathcal{M}} s.ROP_{r-1}.l, \perp)\rangle$.

▶ **Example 2.** Figure 2 shows approximate DDs for the TalentSched instance introduced in Example 1. Despite having a maximum width of 3, the best solution contained in the restricted DD is the optimal solution previously found. With a maximum width of 4, the relaxed DD provides a global lower bound of 86. The path corresponding to this lower bound is given by the assignment $\langle 0, 2, 3, 0\rangle$, which is infeasible because scene 0 is scheduled twice.

2.3.2 Branch-and-Bound

In [9], a branch-and-bound algorithm based only on restricted and relaxed DDs was introduced. It maintains a queue of open nodes that represent the set of subproblems that remain to process. For each of them, a restricted DD is compiled in an attempt to improve the incumbent solution. Then, a relaxed DD is constructed in order to both decompose the given subproblem into even smaller ones and to compute a lower bound for each of them. These nodes are then added to the branch-and-bound queue for further exploration, unless the lower bound permits their direct elimination. Ultimately, the optimality of the best solution discovered during the search is confirmed once the queue has been emptied.

2.3.3 Rough Lower Bound

The *rough lower bound* (RLB) [19] is an additional optional modeling component that can be specified to speed up the resolution of any optimization problem. For any node u, the RLB gives a lower bound on the best value one can obtain when solving the corresponding subproblem $\sigma(u)$, i.e. $\underline{v}_{rlb}(\sigma(u)) \leq v^*(u \rightsquigarrow t \mid \mathcal{B})$ with $\mathcal{B}$ the exact DD for the problem. It is used at line 10 of Algorithm 1 to filter nodes *a priori* by comparing this lower bound with the incumbent value $\overline{v}$. Since the RLB is computed for each node of the approximate DDs compiled throughout the branch-and-bound, it needs to be computationally cheap.

The RLB has the potential both to focus the compilation of restricted DDs on promising parts of the search space and to strengthen the bounds obtained through relaxed DDs. Furthermore, the branch-and-bound algorithm uses the RLB to make pruning decisions, if it happens to be tighter than the bound obtained with relaxed DDs.

Example problems. In our computational experiments, we use the lower bound given by Theorem 1 in [17] for TalentSched and the same RLB as in [20] for the PSP. We do not detail them in this article for the sake of conciseness.

3 Aggregate Dynamic Programming for Decision Diagrams

As stated in the introduction, optimizations techniques based on DP and DDs can prove highly effective [6, 13, 14, 18, 19]. In some cases, however, the state space of the DP models is simply too large and the bounds derived from restricted and relaxed DDs are of little to no use. This can be imputed either to the node selection heuristic or to the relaxation scheme. The *MinLP* heuristic traditionally used favors keeping nodes with the best prefix values. This locally-optimal selection policy may result in the elimination of all nodes that lead to the optimal solution, or even to any feasible solution, particularly in cases of highly constrained problems. In the latter case, the compilation of a restricted DD is a pure waste of time: no feasible solution is found at the end of the compilation, and not even a bound on the objective value can be exploited to reduce the optimality gap. The same phenomenon is detrimental to the usefulness of compiled relaxed DDs whose bounds might be of low quality when the node selection heuristic is oblivious to the global structure of the problem. Indeed, the merging operator yields a loose representation when applied to an arbitrary set of nodes for most problems. In the absence of a perfect heuristic, this situation will occur under certain conditions. It inspired our pursuit of a more globally-focused approach that could enhance the usefulness of the compiled DDs. This section presents a framework for integrating *aggregate dynamic programming* ideas with DD-based optimization that aims to address some of these shortcomings. Instead of relaxing the original problem by reasoning on merged states, it proposes to use problem instance and state aggregation operators that yield a simpler and relaxed version of the problem, which can be solved exactly. Solutions of the aggregated problem can provide bounds that capture the global problem structure, as well as guidance for the compilation of restricted DDs. This section details the role and meaning of the components of the framework one by one.

3.1 Preprocessing: Problem Instance Aggregation

The goal of this preprocessing step is to create an aggregate and simpler problem instance by reducing one or more dimensions of the problem. The *instance aggregation operator* Π must be defined such that the aggregate problem instance $\mathcal{P}' = \Pi(\mathcal{P})$ is a relaxation of the original problem instance $\mathcal{P}$. In practice, assuming the problem reasons over a set of *elements*, a clustering algorithm can be used to create clusters of such elements. Then, the aggregate problem instance can be obtained by considering *aggregate* elements that encompass all elements in a given cluster and by adapting the instance data accordingly. Formally, if a set E of elements is clustered into K clusters, we define two mapping functions: $\Phi : E \to \{0, \ldots, K-1\}$ that gives the cluster for each original element and $\Phi^{-1} : \{0, \ldots, K-1\} \to 2^E$ that gives the set of original elements for a given cluster.

TalentSched. In [17], it is proved that there always exists an optimal solution to the problem in which scenes with the same set of actors are scheduled together. This gives us the opportunity to aggregate the problem by creating K clusters of scenes that require a similar set of actors, which is plausible to occur in real film shoots. Scenes belonging to the same clusters can then be aggregated by taking the intersection of their actor requirements and adding up their durations. Formally, we write $\Pi(\mathcal{P} = (N, A, R, D, C)) = (\Pi_N(N), A, \Pi_R(R), \Pi_D(D), C)$ with $\Pi_N(N) = \{0, \ldots, K-1\}$. The aggregate actor requirements are computed as $\Pi_R(R) = R'$ with $R'_i = \cap_{j \in \Phi^{-1}(i)} R_j$ for all $i \in \Pi_N(N)$ and the aggregate durations as $\Pi_D(D) = D'$ with $D'_i = \sum_{j \in \Phi^{-1}(i)} D_j$ for all $i \in \Pi_N(N)$.

PSP. The number of item types considered in a PSP instance dramatically impacts the size of the state space – for instance, the case with only one item type can be solved greedily. Therefore, and because it is not unlikely that the machine will produce several sets of similar items, we propose to cluster item types that have similar stocking and changeover costs. The instance aggregation operator is thus $\Pi(\mathcal{P} = (I, S, C, H, Q)) = (\Pi_I(I), \Pi_S(S), \Pi_C(C), H, \Pi_Q(Q))$, where the aggregate set of item types is given by $\Pi_I(I) = \{0, \ldots, K-1\}$. Their stocking costs are computed as $\Pi_S(S) = S'$ with $S'_k = \min_{i \in \Phi^{-1}(k)} S_i$ for all $k \in \Pi_I(I)$ and the changeover costs as $\Pi_C(C) = C'$ with $C'_{kl} = \min_{i \in \Phi^{-1}(k), j \in \Phi^{-1}(l)} C_{ij}$ for all $k, l \in \Pi_I(I)$. The aggregate demand matrix is defined as $\Pi_Q(Q) = Q'$ with $Q'^k_p = \sum_{i \in \Phi^{-1}(k)} Q^i_p$. However, as the demand matrix is only supposed to contain unit demands, one must redistribute surplus demands in Q' to the left.

ALP. Similarly to the item types of the PSP, the aircraft classes can be aggregated to reduce the complexity of the problem. We thus propose to cluster them based on their minimum separation time with other classes and define the instance aggregation operator as $\Pi(\mathcal{P} = (N, R, C, A, S, T, L)) = (N, R, \Pi_C(C), \Pi_A(A), \Pi_S(S), T, \Pi_L(L))$. The set of aircrafts, their target landing time and the number of runways is kept. The aggregate set of classes is given by $\Pi_C(C) = \{0, \ldots, K-1\}$ and their corresponding set of aircrafts is computed as $\Pi_A(A) = A'$ with $A'_i = \cup_{j \in \Phi^{-1}(i)} A_j$ for all $i \in \Pi_C(C)$. The smallest separation times between aggregate classes are kept, as formalized by $\Pi_S(S) = S'$ with $S'_{kl} = \min_{i \in \Phi^{-1}(k), j \in \Phi^{-1}(l)} S_{i,j}$ for all $k, l \in \Pi_C(C)$. Finally, the aggregation operator adapts the latest landing times of all the aircrafts so that any aircraft with a given target landing time has a greater latest landing time than all other aircrafts of the same class with a smaller target landing time: $\Pi_L(L) = L'$ with $L'_i = \max\{L_j \mid \Phi(i) = \Phi(j), T_i \leq T_j\}$ for all $i \in A$. This property is assumed to hold for the original problem instance, and must be preserved so that aircrafts from the same class can be scheduled sequentially in the DP model.

▶ **Example 3.** Let us apply the problem instance aggregation to our running example by creating $K = 2$ aggregate scenes. Assuming the following clustering is found: $\Phi(0) = 0, \Phi(1) = 1, \Phi(2) = 0, \Phi(3) = 1$ or equivalently $\Phi^{-1}(0) = \{0, 2\}, \Phi^{-1}(1) = \{1, 3\}$. We thus compute the aggregate scene durations as: $D' = \langle D_0 + D_2, D_1 + D_3 \rangle = \langle 5, 9 \rangle$ and the aggregate actor requirements as: $R' = \langle \{0, 3\} \cap \{0, 2, 3\}, \{0, 1, 3\} \cap \{0, 1, 2, 3\} \rangle = \langle \{0, 3\}, \{0, 1, 3\} \rangle$.

3.2 State Aggregation and Lower Bound

A second mapping function accompanies the problem instance aggregation operator: the *state aggregation operator* $\pi : S \to S'$ that projects each state of the state space S of the original problem in the aggregate state space S'. The role of this operator is to translate each original state to its aggregate version by adapting the state information to fit the aggregate problem data. Let us denote by $\mathcal{B}$ and $\mathcal{B}'$ the exact DD for problem $\mathcal{P}$ and $\Pi(\mathcal{P})$, respectively. If the aggregation operators Π and π are defined such that $v^*(u \rightsquigarrow t \mid \mathcal{B}) \geq v^*(u' \rightsquigarrow t' \mid \mathcal{B}')$ for all $u \in \mathcal{B}, u' \in \mathcal{B}'$ with $\pi(\sigma(u)) = \sigma(u')$ and $\pi(\sigma(t)) = \sigma(t')$, then $v^*(u' \rightsquigarrow t' \mid \mathcal{B}')$ can be used as a lower bound in the original problem, which we will denote by $\underline{v}_{agg}(\pi(\sigma(u)))$.

Assuming the aggregate problem can be pre-solved exactly and the solution of each subproblem is stored, this aggregation-based lower bound can be retrieved very quickly. One way to exploit it is to incorporate it in the RLB as shown at line 9 of Algorithm 1 so that it is used as often as possible. Another possibility would be to use the aggregate state space to replace the state merging scheme in relaxed DDs. Once a layer with greater width than W is reached, all the states contained in the nodes of the layer could be mapped to the aggregate state space to pursue the compilation in a lower dimensional space.

TalentSched. The state compression operator for TalentSched is somewhat complex because we can only map to states where complete aggregate scenes have yet to be scheduled. As a result, if a state s contains scenes in $s.P$ that can optionally be scheduled, we map it to a dummy aggregated state. The same logic is applied when $s.M$ only contains a subset of the scenes that compose an aggregate scene.

$$\pi(s) = \begin{cases} (\emptyset, \emptyset), & \text{if } s.P \neq \emptyset, \\ (\emptyset, \emptyset), & \text{if } \exists i \in \Pi_N(N) : (\Phi^{-1}(i) \cap s.M) \neq \emptyset \wedge \Phi^{-1}(i) \nsubseteq s.M, \\ (M', \emptyset), & \text{otherwise, with } M' = \left\{ i \in \Pi_N(N) \mid \Phi^{-1}(i) \subseteq s.M \right\}. \end{cases}$$

PSP. If we extend the definition of Φ such that $\Phi(\bot) = \bot$, the state aggregation operator can be defined as $\pi(s) = (\Phi(s.i), R)$ with $R_i = \sum_{j \in \Phi^{-1}(i)} s.R_j$ for all $i \in \Pi_I(I)$. The item type is projected to its corresponding aggregate type, and the remaining number of items to produce for each type is separately accumulated within each cluster.

ALP. Again, assuming $\Phi(\bot) = \bot$, the state aggregation operator is defined by $\pi(s) = (Q', ROP')$ with the remaining quantities of aircrafts aggregated as $Q'_i = \sum_{j \in \Phi^{-1}(i)} s.Q_j$ for all $i \in \Pi_C(C)$. For the ROP, one only needs to adapt the class of the last aircraft scheduled on each runway $ROP'_i = (s.ROP_0.l, \Phi(s.ROP_0.c))$ for all $i \in R$.

If lower bounds for original states are obtained only by pre-solving the aggregate problem, it is unlikely that the solution of an aggregate subproblem mapped with the state aggregation operator will be available, since the aggregate separation times between aircraft classes lead to very different landing times. However, a lower bound for an aggregate state $s^1 = (Q^1, ROP^1)$ can be provided by the solution of any state $s^2 = (Q^2, ROP^2)$ such that $Q^1 = Q^2$ and $ROP^1_i.c = ROP^2_i.c$ and $ROP^1_i.l \geq ROP^2_i.l$ for all $i \in R$.

▶ **Example 4.** Let us compute the aggregation-based lower bound for the root state of the running example $r = (\{0, 1, 2, 3\}, \emptyset)$ given its aggregate version $\pi(r) = (\{0, 1\}, \emptyset)$ and the clustering performed in Example 3. The aggregate version is trivial to solve since the objective function is symmetrical and there are only two scenes to schedule. We thus have $\underline{v}_{agg}(r) = D'_0 \times (C_0 + C_3) + D'_1 \times (C_0 + C_1 + C_3) = 5 \times (1 + 4) + 9 \times (1 + 2 + 4) = 88$, which is a slightly better lower bound than the one obtained with the relaxed DD of Example 2.

3.3 Solution Disaggregation and Node Selection Heuristic

In order to exploit the solution of the aggregate version of a subproblem to find good heuristic solutions for the original subproblem, we need to specify the correspondence between decisions in the aggregate problem with decisions in the original problem. We therefore define a last modeling component, called the *decision disaggregation operator* $\delta(d) : D'_k \to 2^{D_i} \times \cdots \times 2^{D_j}$ that maps the instantiation of a variable x'_k in the aggregate problem to a vector of possible corresponding assignments for variables $x_i, \ldots, x_j$ in the original problem.

Finally, we define the *path disaggregation operator* that transforms a sequence of decisions in the aggregate problem to a sequence of sets of possible decisions in the original problem: $\Delta(p = (a_k, \ldots, a_{n'-1})) = \delta(l(a_k)) \cdot \ldots \cdot \delta(l(a_{n'-1}))$ where n' is the supposed number of aggregate variables and $\cdot$ denotes the *concatenation* of two vectors. Using this operator, we can compute a *score* for each decision made during the compilation of restricted DDs. At line 3 of Algorithm 1, we first retrieve the optimal value assignment of the aggregate subproblem and apply the path disaggregation operator on it. Then, a binary *score* is attributed to each arc at line 15, depending on its compatibility with the disaggregated solution. At line 2 of Algorithm 2, the maximum score obtained along any path up to each node can then be used to order nodes from most to least promising, favoring nodes with incoming paths that are highly compatible with the disaggregated solution. By doing so, the width of restricted DDs is controlled in the same way as before, enabling the preference of solutions even when no feasible solution with the maximum possible score is available.

TalentSched. Each aggregate scene corresponds to a set of original scenes, we thus need to map each aggregate decision to a sequence of original decisions: $\delta(i) = V$ where $V_j = \Phi^{-1}(i)$ for all $0 \leq j < |\Phi^{-1}(i)|$. It corresponds to any of the scenes from the cluster i, duplicated $|\Phi^{-1}(i)|$ times so that they are all scheduled one after another, preferably.

PSP. The operator is much simpler to define for the PSP, since each decision concerns the production of one unit of a chosen aggregate item type. It can thus be interpreted as the decision of producing one unit of any item type in the corresponding cluster: $\delta(i) = \langle \Phi^{-1}(i) \rangle$.

ALP. The only difference with the PSP is that decisions also contain the runway on which the aircraft is scheduled to land, which remains the same: $\delta(a, r) = \langle \{(a', r) \mid a' \in \Phi^{-1}(a)\} \rangle$.

▶ **Example 5.** As computed in Example 4, the schedule $\langle 0, 1 \rangle$ is optimal for the aggregate problem. By disaggregating this solution, we get $\langle \{0, 2\}, \{0, 2\}, \{1, 3\}, \{1, 3\} \rangle$. We can notice that the optimal schedule $\langle 0, 2, 3, 1 \rangle$ found in Example 1 is compatible with the disaggregated solution and would thus be favored by the aggregation-based node selection heuristic.

4 Computational Experiments

The impact of the aggregation-based bounds and heuristics was evaluated experimentally by extending the generic DD-based solver DDO [21] and injecting the modeling of the three discrete optimization problems presented throughout the paper. The version of DDO used includes the improvements introduced in [16, 19]. For each problem, random instances were generated with the following main parameters:

- TalentSched: number of scenes $n \in \{20, 22, 24, 26, 28\}$, number of actors $m \in \{10, 15\}$ and average fraction of actors required for each scene $\rho \in \{0.3, 0.4\}$.

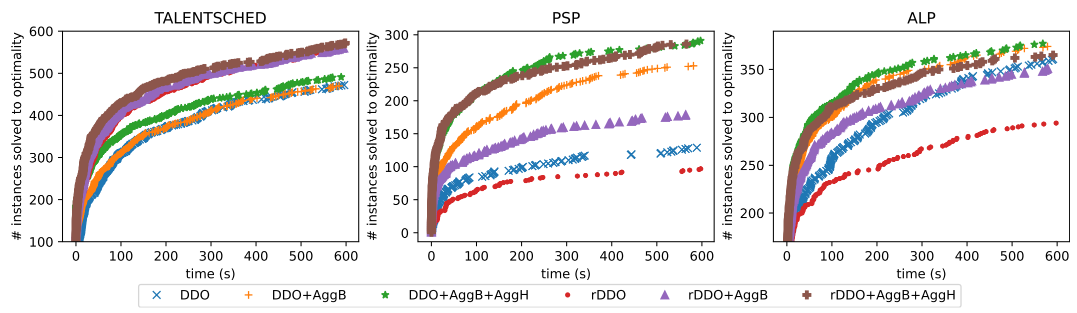

Figure 3 Number of instances solved over time for each configuration and problem studied.

- PSP: number of item types $n = 10$, horizon $H \in \{100, 150, 200\}$ and fraction of time periods with a demand $\rho \in \{0.9, 0.95, 1\}$.
- ALP: number of aircrafts $n \in \{25, 50, 75, 100\}$, number of runways $r \in \{1, 2, 3, 4\}$, number of aircraft classes $c = 4$ and mean inter-arrival time $40/r$ for generating the target landing times according to a Poisson arrival process.

Furthermore, the instance generation tries to emulate an increasing number of groups of actor requirements, item types and aircraft classes that lend themselves more or less to aggregation. Each instance was presolved in its aggregate state space after aggregating its data according to k-means clustering for PSP and ALP and a custom hierarchical clustering for TalentSched that tries to maximize the remaining costs induced by the actor requirements. TalentSched instances can be presolved exactly with 20 aggregate scenes and PSP instances similarly with 5 aggregate item types. On the other hand, not all ALP instances reduced to 2 aggregate aircraft have a reasonable number of states so we employ a relaxed DD with maximum width 10000 for the presolving part instead. Note that the present approach does not compete with the state-of-the-art for TalentSched as it lacks much of the custom symmetry-breaking logic introduced in [17] and similarly for ALP regarding the dominance-breaking constraints presented in [28]. Six different configurations were created by combining the default DD-based solved DDO on one hand and a version using only restricted DDs and no relaxed DDs, denoted rDDO, on the other hand, with the aggregation-based bounds (AggB) and heuristics (AggH). Ten minutes were allotted for each configuration to solve each instance.

Figure 3 presents the cumulative number of instances solved with respect to the solving time. For TalentSched, it appears that any configuration of rDDO performs better than any of DDO. This suggests that the bounds provided by the relaxed DDs are looser than the RLB while being more expensive to compute. It confirms our intuition that the state merging scheme yields bounds with a limited impact for some problems, probably because the state information gets very dilute when many states are merged together. In this case, the RLB computation is also quite involved – see [17]. Still, adding the AggB and the AggH to either configurations improves the results by a small margin, although not that significant. This can be contrasted with the results obtained for the two other problems, which show a clear improvement when the AggB and the AggH are added to either configurations. Furthermore, in cases where rDDO alone yields the worst results, incorporating AggB leads to results that are similar to or better than those achieved by DDO. Combining it with the AggH performs better than DDO in both cases and almost equally well than DDO+AggB+AggH.

The impact of the AggB and the AggH can also be measured in terms of end gap $\frac{UB-LB}{UB}$. Figure 4 compares the end gap obtained for each instance by DDO and DDO+AggB+AggH. It shows that except for a few instances, DDO+AggB+AggH is always closer to terminating the search than DDO, especially for PSP. To validate the relevance of the AggH, we also

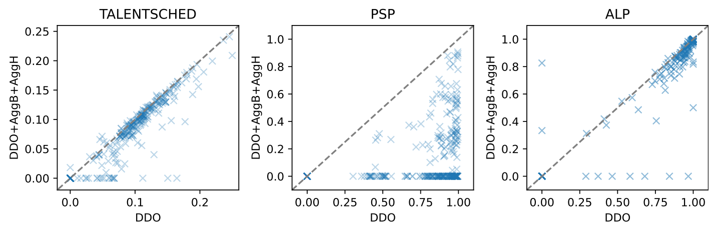

Figure 4 Comparison of the end gap obtained for each instance by DDO and DDO+AggB+AggH.

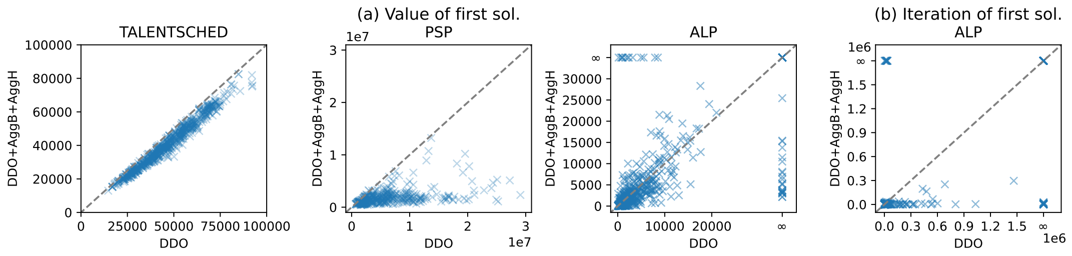

Figure 5 Comparison of the value of the first solution found by DDO and DDO+AggB+AggH, and of the iteration at which the solution is found for ALP.

compare the value of the first solution found by DDO and DDO+AggB+AggH on Figure 5(a). For TalentSched and PSP, the quality of the first solution is always better when using the AggH. However, there is no clear trend for the ALP. Unlike TalentSched and PSP, for which a solution is always found at the first iteration, the landing time windows of ALP make it difficult to find a feasible solution. This explains both the end gaps close to one in Figure 4 and the ∞ values in Figure 5(a), which represent the absence of a feasible solution. We thus compare on Figure 5(b) the iteration at which the first solution is found. We observe that DDO+AggB+AggH finds a feasible solution much earlier than DDO in most cases. This showcases well the benefits of a node selection heuristic with a more global awareness.

5 Conclusion

This paper explained how ideas from aggregate dynamic programming can be incorporated in DD-based optimization solvers. We proposed to derive lower bounds and node selection heuristics from a pre-solved aggregate version of the original problem at hand, and explained how these can be seamlessly added to the DD-based optimization framework. Computational experiments on three different problems showed that they provide lower bounds that further strengthen the current approach, and that could even be used as a replacement for relaxed DDs in some cases. Furthermore, the aggregation-based node selection heuristics were shown very valuable as they manage to steer the compilation of relaxed DDs toward better solutions earlier in the search. When applying this idea to a highly constrained problem, the heuristics proved to quickly lead to feasible solutions that were hard to find otherwise. These results suggest that aggregation-based bounds and heuristics capture global problem structures well, as opposed to the greedy *MinLP* heuristic traditionally used to compile approximate DDs.

References

1 Henrik Reif Andersen, Tarik Hadzic, John N Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.

2 Sven Axsäter. State aggregation in dynamic programming—an application to scheduling of independent jobs on parallel processors. *Operations Research Letters*, 2(4):171–176, 1983.

3 James C Bean, John R Birge, and Robert L Smith. Aggregation in dynamic programming. *Operations Research*, 35(2):215–220, 1987.

4 Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, November 1954. URL: `https://projecteuclid.org:443/euclid.bams/1183519147`.

5 David Bergman and Andre A. Cire. On finding the optimal bdd relaxation. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, volume 10335 of *LNCS*, pages 41–50. Springer, 2017.

6 David Bergman, Andre A Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat, and Willem-Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In *Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings 11*, pages 351–367. Springer, 2014.

7 David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John N Hooker. 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*, pages 34–49. Springer, 2012.

8 David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John N Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26(2):253–268, 2014.

9 David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John N Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

10 David Bergman, Andre A Cire, Willem-Jan van Hoeve, and Tallys Yunes. Bdd-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, 2014.

11 Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 34(5):2552–2570, 2022.

12 Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

13 Margarita P Castro, Andre A Cire, and J Christopher Beck. An mdd-based lagrangian approach to the multicommodity pickup-and-delivery tsp. *INFORMS Journal on Computing*, 32(2):263–278, 2020.

14 Margarita P Castro, Chiara Piacentini, Andre Augusto Cire, and J Christopher Beck. Solving delete free planning with relaxed decision diagram based heuristics. *Journal of Artificial Intelligence Research*, 67:607–651, 2020.

15 Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

16 Vianney Coppé, Xavier Gillard, and Pierre Schaus. Decision diagram-based branch-and-bound with caching for dominance and suboptimality detection, 2023. `arXiv:2211.13118`.

17 Maria Garcia de la Banda, Peter J Stuckey, and Geoffrey Chu. Solving talent scheduling with dynamic programming. *INFORMS Journal on Computing*, 23(1):120–137, 2011.

18 Xavier Gillard. *Discrete optimization with decision diagrams: design of a generic solver, improved bounding techniques, and discovery of good feasible solutions with large neighborhood search*. PhD thesis, UCL-Université Catholique de Louvain, 2022.

19 Xavier Gillard, Vianney Coppé, Pierre Schaus, and André Augusto Cire. Improving the filtering of branch-and-bound mdd solver. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 231–247. Springer, 2021.

20 Xavier Gillard and Pierre Schaus. Large neighborhood search with decision diagrams. In *International Joint Conference on Artificial Intelligence*, 2022.

21 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for mdd-based optimization. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 5243–5245, 2021.

22 Jaime E Gonzalez, Andre A Cire, Andrea Lodi, and Louis-Martin Rousseau. Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem. *Constraints*, pages 1–24, 2020.

23 John N. Hooker. Improved job sequencing bounds from decision diagrams. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, volume 11802 of *LNCS*, pages 268–283. Springer, 2019.

24 Matthias Horn, Johannes Maschler, Günther R Raidl, and Elina Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research*, 126:105125, 2021.

25 Alan J. Hu. *Techniques for efficient formal verification using binary decision diagrams*. PhD thesis, Stanford University, Department of Computer Science, 1995.

26 Anthony Karahalios and Willem-Jan van Hoeve. Variable ordering for decision diagrams: A portfolio approach. *Constraints*, 27(1):116–133, 2022.

27 C.-Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

28 Alexander Lieder, Dirk Briskorn, and Raik Stolletz. A dynamic programming approach for the aircraft landing problem with aircraft classes. *European Journal of Operational Research*, 243(1):61–69, 2015.

29 Shin-ichi Minato. *Binary decision diagrams and applications for VLSI CAD*, volume 342. Springer Science & Business Media, 1995.

30 Isaac Rudich, Quentin Cappart, and Louis-Martin Rousseau. Peel-And-Bound: Generating Stronger Relaxed Bounds with Multivalued Decision Diagrams. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

31 Christian Tjandraatmadja. *Decision Diagram Relaxations for Integer Programming*. PhD thesis, Carnegie Mellon University Tepper School of Business, 2018.

32 Christian Tjandraatmadja and Willem-Jan van Hoeve. Target cuts from relaxed decision diagrams. *INFORMS Journal on Computing*, 31(2):285–301, 2019. `doi:10.1287/ijoc.2018.0830`.