Modeling and Exploiting Dominance Rules for Discrete Optimization with Decision Diagrams

Vianney Coppé[0000−0001−5050−0001], Xavier Gillard[0000−0002−4493−6041], and Pierre Schaus[0000−0002−3153−8941]

UCLouvain, Louvain-la-Neuve, Belgium
{vianney.coppe,xavier.gillard,pierre.schaus}@uclouvain.be

Abstract. Discrete optimization with decision diagrams is a recent approach to solve combinatorial problems that can be formulated with dynamic programming. It consists in a branch-and-bound algorithm that iteratively explores the search space by compiling bounded-width decision diagrams. Those decision diagrams are used both to subdivide a given problem into smaller subproblems – in a divide-and-conquer fashion – and to compute primal and dual bounds for those. It has been previously shown that pruning performed during the compilation of those decision diagrams can greatly impact the quality of the bounds, and consequently the performance of the branch-and-bound algorithm. In this paper, we study the integration of dominance rules inside the decision diagram-based optimization framework. We propose a modeling language for consistently formulating dominance rules for dynamic programming models, and describe how they can be exploited to systematically detect and prune dominated nodes during the search. Furthermore, we explain how to combine this additional filtering mechanism with caching techniques to further improve the performance of the algorithm. Dominance rules are shown to significantly reduce the number of nodes expanded and the running time of the algorithm on four optimization problems.

Keywords: Decision diagrams · Branch-and-bound · Dominance rules.

1 Introduction

Discrete optimization with *decision diagrams* (DDs) [3] is a recent framework for solving *dynamic programming* formulations of discrete optimization problems through *branch-and-bound* (B&B). It relies on bounded-width DDs to subdivide the problem into smaller subproblems and compute bounds for those. In particular, the compilation of restricted DDs generates feasible solutions in a *beam search* fashion. Inversely, relaxed DDs automatically compute dual bounds by means of a problem-specific state merging operator. As shown in [9,14], filtering techniques that prune nodes *a priori* during the top-down compilation of approximate DDs can greatly impact the performance of the B&B algorithm. On the one hand, restricted DDs produce better solutions because they are guided towards promising parts of the search space. On the other hand, the pruning

performed inside relaxed DDs further shrinks the areas of the search space that effectively needs to be explored, which facilitates the work of the B&B algorithm.

Dominance rules are another well-known ingredient that can reduce the size of the search tree by filtering subproblems leading to redundant solutions. They were first formalized in [19,20] in the general case of a B&B framework. Several optimization paradigms successfully applied them, including MIP [12], CP [8,23,25] and DP [4,7,16,31]. In any of those technologies, dominance rules play a crucial role in facilitating the solving process when applicable. Therefore, it is a very natural step to incorporate this ingredient inside DD-based B&B solvers. This paper is, to the best of our knowledge, the first to fill this gap for this particular field of research, although similar work has already been done for the neighboring line of research on state space search for optimization [21]. After a brief summary of the DD-based B&B algorithm in Section 2, it starts by providing general definitions about dominance rules within the context of DD-based optimization in Section 3, and describe how dominance rules can be formulated for DP models. Section 4 then explains how they can be exploited to systematically detect and prune dominated nodes during the search. Next, a brief explanation on how to combine this additional filtering mechanism with the caching techniques proposed in [9] is given in Section 5. Finally, we present in Section 6 the experimental evaluation of the integration of dominance rules for four different optimization problems, and discuss the results before concluding.

2 Preliminaries

Dynamic Programming The DD-based optimization framework introduced in [3] manipulates a discrete optimization problem $\mathcal{P}$ through a DP model composed of the following elements:

- a vector of *control variables* $x = (x_0, \dots, x_{n-1})$ with $x \in \mathcal{D} = \mathcal{D}_0 \times \cdots \times \mathcal{D}_{n-1}$ and $x_j \in \mathcal{D}_j$ for each $j \in \{0, \dots, n-1\}$.
- a *state space* $\mathcal{S}$ partitioned into $n+1$ sets $\mathcal{S}_0, \dots, \mathcal{S}_n$ corresponding to the successive stages of the DP model. In particular, we define the *root* – or *initial* – state $\hat{r}$, the *terminal* state $\hat{t}$ and the *infeasible* state $\hat{0}$.
- a set of *transition functions* $t_j : \mathcal{S}_j \times \mathcal{D}_j \to \mathcal{S}_{j+1}$ with $j = 0, \dots, n-1$ encoding the transition from one state s^j to another s^{j+1}, according to the decision d made about variable x_j.
- a set of *transition value functions* $h_j : \mathcal{S}_j \times \mathcal{D}_j \to \mathbb{R}$ that specify the reward of assigning some value $d \in \mathcal{D}_j$ to the variable x_j for each $j = 0, \dots, n-1$.
- a *root value* v_r to model constant terms in the objective.

Given such a DP model, the optimal solution can be obtained by solving:

$$\begin{aligned} \text{maximize } f(x) &= v_r + \sum_{j=0}^{n-1} h_j(s^j, x_j) \\ \text{subject to } s^{j+1} &= t_j(s^j, x_j), \text{ for all } j = 0, \dots, n-1, \text{ with } x_j \in \mathcal{D}_j \\ s^j &\in \mathcal{S}_j, s^j \neq \hat{0}, j = 0, \dots, n. \end{aligned}$$

Decision Diagrams DDs are used in a variety of domains to compactly encode a set of solutions, and that also applies to those induced by a DP model. With this specific application in mind, a DD $\mathcal{B} = (U, A, \sigma, l, v)$ is defined as a layered directed acyclic graph with U the set of nodes and A the set of arcs. The state function σ maps each node $u \in U$ to a DP state $\sigma(u) \in \mathcal{S}$. The set of nodes U is partitioned into a set of *layers* $L_0, \dots, L_n$ that correspond to stages of the DP model. Each transition between pairs of states $s^j \in \mathcal{S}_j$ and $s^{j+1} \in \mathcal{S}_{j+1}$ is materialized by an arc $a = (u_j \xrightarrow{d} u_{j+1})$ that connects the corresponding nodes $u_j \in L_j, u_{j+1} \in L_{j+1}$, with $\sigma(u_j) = s^j$ and $\sigma(u_{j+1}) = s^{j+1}$. The *label* $l(a) = d$ of each arc represents the assignment of decision $d \in \mathcal{D}_j$ to variable x_j, and its value $v(a)$ captures the transition value. Both the first and last layer – L_0 and L_n – contain a single node, respectively the *root* r and the *terminal* node t. Consequently, each $r \rightsquigarrow t$ path $p = (a_0, \dots, a_{n-1})$ that connects the root and the terminal node through the arcs $a_0, \dots, a_{n-1}$ encodes a solution $x(p) = (l(a_0), \dots, l(a_{n-1}))$ with value $v(p) = v_r + \sum_{j=0}^{n-1} v(a_j)$. DD $\mathcal{B}$ is said *exact* if it exactly represents the set of solutions of the corresponding problem, i.e. $Sol(\mathcal{B}) = Sol(\mathcal{P})$ and $v(p) = f(x(p)), \forall p \in \mathcal{B}$, with $Sol(\mathcal{B}) = \{x(p) \mid \exists p : r \rightsquigarrow t, p \in \mathcal{B}\}$. We denote by $v^*(u \mid \mathcal{B})$ the value of the longest path that reaches node u within a DD $\mathcal{B}$, and define $v^*(\mathcal{B}) = v^*(t \mid \mathcal{B})$ for conciseness.

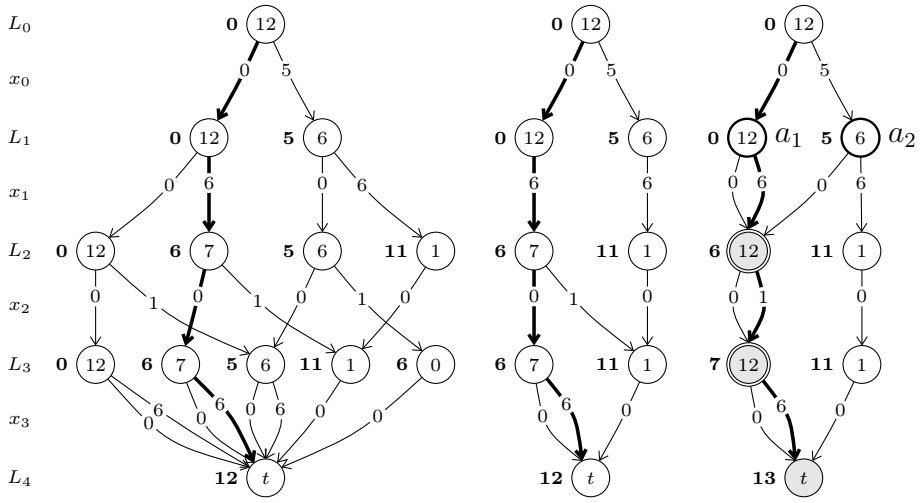
Example 1. Given a set of items $N = \{0, \dots, n-1\}$, along with their weights $W = \langle w_0, \dots, w_{n-1} \rangle$ and values $V = \langle v_0, \dots, v_{n-1} \rangle$, the goal of the *0–1 Knapsack Problem* (KP) is to select a subset of items that maximizes the total value while keeping the total weight under a given capacity C. In its well-known DP formulation, each item $j \in N$ is associated with a binary variable x_j that decides whether to include it in the knapsack. States simply contain the remaining capacity of the knapsack. The state space is thus defined as $S = [0, C]$, with the root state $\hat{r} = C$ starting at maximum capacity, and with root value $v_r = 0$. The transition functions are given by:

$$t_j(s^j, x_j) = \begin{cases} s^j - x_j w_j, & \text{if } x_j w_j \leq s^j, \\ \hat{0}, & \text{otherwise,} \end{cases}$$

meaning that the weight of item j is subtracted from the remaining capacity when it is added to the knapsack. If the capacity constraint is violated, the transition is redirected to the infeasible state. Likewise, the transition value functions $h_j(s^j, x_j) = x_j v_j$ add the value of item j if it is included in the knapsack.

Let us consider an instance of the KP with $n = 4$, $C = 12$, $W = \langle 6, 5, 6, 6 \rangle$ and $V = \langle 5, 6, 1, 6 \rangle$. Figure 1(a) shows the exact DD for that problem. The longest path corresponds to the optimal solution $x^*(\mathcal{B}) = (0, 1, 0, 1)$ for a value of $v^*(\mathcal{B}) = 12$ and a total weight of 11.

DD compilation Algorithm 1 describes the top-down compilation of a DD $\mathcal{B}$ for a given DP model. It takes a root node u_r and a maximum width W as input and recursively builds the DD by applying all valid transitions to the last completed layer. When the number of nodes in the last completed layer exceeds



(a) Exact (b) Restricted (c) Relaxed

Fig. 1. Exact, restricted and relaxed DDs for the KP instance of Example 1. The value inside each node u corresponds to its state $\sigma(u)$ – the remaining capacity – and the annotation on the left gives the value of the longest path that reaches it $v^*(u \mid \mathcal{B})$. For clarity, only arc values are present. The longest path is highlighted in bold.

the parameter W at line 7, the algorithm compiles an *approximate* DD, as will be detailed next. To create the next layer, the algorithm iterates over all nodes of the last completed layer and applies all valid DP transitions to them at lines 10 to 10 before encoding them as arcs and nodes. Note that a single node is created for each *distinct* state reached by the transitions. The last step of the algorithm is to merge all nodes of the terminal layer into a single terminal node t at line 15.

Approximate DDs When the size of a layer exceeds the parameter W at line 7 of Algorithm 1, two procedures exist to reduce the number of nodes in the layer. Restricted DDs adopt a simple strategy that consists in heuristically removing the least promising nodes of the layer, as described by Algorithm 2. They thus produce a subset of the solutions of the problem, which provide lower bounds on the optimal solution. For a restricted DD $\underline{\mathcal{B}}$, we thus have that $Sol(\underline{\mathcal{B}}) \subseteq Sol(\mathcal{P})$ and $v(p) = f(x(p)), \forall p \in \underline{\mathcal{B}}$. On the other hand, relaxed DDs over-approximate the set of solutions of the problem by locally relaxing the problem by *merging* surplus nodes together. To this end, state merging operators ensuring that no feasible solutions are removed must be defined for each DP model. If $\mathcal{M}$ is the set of nodes to merge and $\sigma(\mathcal{M}) = \{\sigma(u) \mid u \in \mathcal{M}\}$ the corresponding set of states, the operator $\oplus(\sigma(\mathcal{M}))$ gives the state of the merged node. In Algorithm 2, this operator is used at line 4 to create a single *meta*-node and at lines 5 to 6, the arcs pointing to the merged nodes are redirected to it. A second operator denoted

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

1: $i \leftarrow$ index of the layer containing u_r
2: $L_i \leftarrow \{u_r\}$
3: **for** $j = i$ **to** $n-1$ **do**
4: $\quad pruned \leftarrow \emptyset$
5: $\quad$ perform dominance pruning using Algorithm 3
6: $\quad L'_j \leftarrow L_j \setminus pruned$
7: $\quad$ **if** $|L'_j| > W$ **then**
8: $\quad\quad$ restrict or relax the layer to get W nodes with Algorithm 2
9: $\quad L_{j+1} \leftarrow \emptyset$
10: $\quad$ **for all** $u \in L'_j$ **do**
11: $\quad\quad$ **for all** $d \in \mathcal{D}_j$ **do**
12: $\quad\quad\quad$ create node u' with state $\sigma(u') = t_j(\sigma(u), d)$ or retrieve it from L_{j+1}
13: $\quad\quad\quad$ create arc $a = (u \xrightarrow{d} u')$ with $v(a) = h_j(\sigma(u), d)$ and $l(a) = d$
14: $\quad\quad\quad$ add u' to L_{j+1} and add a to A
15: merge nodes in L_n into terminal node t

Algorithm 2 Restriction or relaxation of layer L'_j with maximum width W.

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

$\Gamma_{\mathcal{M}}$ can be defined to adjust the value of the arcs incident to the merged node at line 6. For valid relaxation operators, a relaxed DD $\overline{\mathcal{B}}$ verifies $Sol(\overline{\mathcal{B}}) \supseteq Sol(\mathcal{P})$ and $v(p) \geq f(x(p)), \forall p \in \overline{\mathcal{B}}$. Whereas restricted DDs only contain *exact* nodes, relaxed DDs also contain *relaxed* nodes that are either merged nodes, or nodes that are reached by at least one path that traverses a merged node.

Branch-and-bound The B&B algorithm introduced in [3] builds upon those two types of approximate DDs to solve problems to optimality. Restricted DDs are used to generate feasible solutions from any B&B node, while relaxed DDs decompose the problem further and provide dual bounds for the subproblems thus created. Indeed, in a relaxed DD, it is possible to identify a set of exact nodes whose associated subproblems collectively represent the root problem, and therefore solving them is equivalent to solving the root problem. The B&B algorithm maintains a queue of such nodes to process, and uses restricted DDs to try to improve the best solution found so far, and relaxed DDs to further decompose the problem and prune unpromising subproblems. When the algorithm terminates, all solutions have been either enumerated or pruned. For the sake of conciseness, we do not detail here the additional filtering techniques that have been proposed in [9,14] to speed up this process.

Example 2. Figure 1(b) shows the result of compiling a restricted DD with maximum width $W = 2$, for the KP instance of Example 1. By applying the greedy heuristic that deletes nodes with the lowest prefix values, the best solution that the restricted DD obtains is $x^*(\underline{\mathcal{B}}) = (0, 1, 0, 1)$ with a value of $v^*(\underline{\mathcal{B}}) = 12$. This lower bound is actually the optimal solution to the problem.

To compile a relaxed DD for this problem, we first define a state merging operator: $\oplus(\sigma(\mathcal{M})) = \max_{s \in \sigma(\mathcal{M})} s$, which keeps the maximum remaining capacity among the states to merge. The operator $\Gamma_{\mathcal{M}}$ is the identity function here since there is no need to modify the arc values. Given those operators, Figure 1(c) shows a relaxed DD compiled with $W = 2$. The longest path in this diagram corresponds to the solution $x^*(\overline{\mathcal{B}}) = (0, 1, 1, 1)$ for a value of $v^*(\overline{\mathcal{B}}) = 13$ and a total weight of 17. This solution violates the capacity constraint, which can happen because the state merging operator relaxes this constraint. Nevertheless, it provides an upper bound for the problem. If we were to solve the problem to optimality, a set of exact nodes to explore next would be extracted from the relaxed DD. For instance, nodes a_1 and a_2 could be added to the B&B queue.

3 Dominance Rules for Decision Diagrams

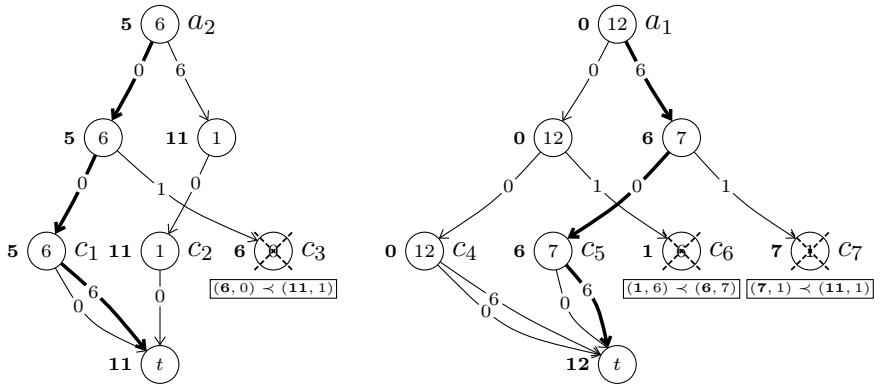
Let us now define the concept of node dominance in the DD-based optimization context. It only concerns exact nodes because relaxed nodes have both a relaxed value and state representation, and thus do not produce valid dominance relations. In the following, the operator $\cdot$ denotes the concatenation of two vectors.

Definition 1 (Node Dominance). *Let $u_1 \in \mathcal{B}_1$ and $u_2 \in \mathcal{B}_2$ be two exact nodes respectively obtained in DDs $\mathcal{B}_1$ and $\mathcal{B}_2$ compiled for a problem $\mathcal{P}$, and whose states belong to the j-th stage of the corresponding DP model, meaning that $\sigma(u_1), \sigma(u_2) \in \mathcal{S}_j$. We say that u_1 dominates u_2 – written as $u_1 \succ u_2$ – if for any partial assignment $(x_j, \ldots, x_{n-1}) \in \mathcal{D}_j \times \cdots \times \mathcal{D}_{n-1}$ such that $x_2 = x^*(u_2 \mid \mathcal{B}_2) \cdot (x_j, \ldots, x_{n-1}) \in Sol(\mathcal{P})$, we also have that $x_1 = x^*(u_1 \mid \mathcal{B}_1) \cdot (x_j, \ldots, x_{n-1}) \in Sol(\mathcal{P})$ and either:*

- *$\sigma(u_1) \neq \sigma(u_2)$ and $f(x_1) \geq f(x_2)$,*
- *or, $\sigma(u_1) = \sigma(u_2)$ and $f(x_1) > f(x_2)$.*

If we are interested in finding a single optimal solution to the problem and that such dominance relation exists between nodes u_1 and u_2, then clearly the exploration of node u_2 can be avoided. For some DP models, *dominance rules* that systematically identify scenarios where this kind of node dominance relation exists can be derived. That is, they provide a simple criterion to detect dominated nodes without needing to expand them in the first place and determine algorithmically whether such dominance relation arises. We define such dominance rules through two components:

- The *dominance key* operator $\kappa : \mathcal{S} \to \mathcal{S}'$ that maps each state of the state space $\mathcal{S}$ of a DP model to a *reduced* state in a *reduced* state space $\mathcal{S}'$. This operator partitions the state space $\mathcal{S}$ in equivalence classes $\mathcal{S}^0, \ldots, \mathcal{S}^M$ such



(a) Exact DD from a_2 (b) Exact DD from a_1

Fig. 2. Exact DD compiled from nodes a_1 and a_2 of Figure 1(b) and exploiting the dominance rule for this problem.

that $\forall s_1, s_2 \in \mathcal{S}^m : \kappa(s_1) = \kappa(s_2)$ for all $m = 0, \ldots, M$. The dominance key typically contains a subset of the original state definition and the equivalence classes group states that are eligible for a dominance relation.

– Furthermore, the *partial dominance utility* operator $\psi : \mathcal{S} \to \mathbb{R}^k$ transforms each state into a vector of k coordinates. Given a node $u \in \mathcal{B}$, we also define the *dominance utility* operator $\Psi(u) = (v^*(u \mid \mathcal{B})) \cdot \psi(\sigma(u))$ that concatenates the node value with the partial utility vector, producing a vector in $\mathbb{R}^{k+1}$ that must characterize the utility of the corresponding node.

These modeling ingredients are similar, yet slightly more flexible than the resource-based approach adopted in [21], since they allow reasoning over quantities other than state variables. The following definition formalizes the connection between those operators and Definition 1, and the necessary condition for those modeling components to constitute a valid dominance rule. It assumes that, given two vectors $x, y \in \mathbb{R}^{k+1}$, we write $x \geq y$ if $x_i \geq y_i$ for $i = 0, \ldots, k$ and $x \neq y$.

Definition 2 (Dominance Rule). *The operators κ and ψ define a valid dominance rule for a given DP model if, for any two exact nodes $u_1 \in \mathcal{B}_1$ and $u_2 \in \mathcal{B}_2$ obtained in the j-th layer of DDs $\mathcal{B}_1$ and $\mathcal{B}_2$, having $\kappa(\sigma(u_1)) = \kappa(\sigma(u_2))$ and $\Psi(u_1) \geq \Psi(u_2)$ implies that $u_1 \succ u_2$ holds.*

Example 3. In the case of the KP, a node u_1 having both higher value and remaining capacity than another node u_2 will always produce better solutions. This dominance rule can be formulated through the following dominance key: $\kappa(s) = \mathbf{0}$ for each state s, which is the zero-dimensional vector, since all states of the same stage can be compared. The partial dominance utility is simply $\psi(s) = s$ so that for a node $u \in \mathcal{B}$, the dominance utility operator compares the value and the remaining capacity $\Psi(u) = (v^*(u \mid \mathcal{B})) \cdot \psi(\sigma(u)) = (v^*(u \mid \mathcal{B}), \sigma(u))$. Figure 2(a)

shows a DD compiled from node a_2 of Figure 1(c), obtained by performing dominance checks using the rule defined above during the compilation. Node c_3 is dominated by c_2 since $\kappa(c_3) = \kappa(c_2) = \mathbf{0}$ and $\Psi(c_3) = (6, 0) \leq \Psi(c_2) = (11, 1)$. It can thus be pruned, resulting in an exact DD even with $W = 2$.

4 Filtering the Search Using Dominance Rules

In this section, we explain how to systematically detect and prune dominated nodes within the DD-based B&B algorithm, based on the modeling ingredients previously defined. In this regard, we propose two strategies that both fall in the category of *memory-based dominance relations* [26].

- The first is to perform dominance checks exclusively for nodes belonging to the same layer of the same DD. This way, no extra memory is required and the number of nodes for which dominance relations are checked is kept small.
- On the other hand, the second strategy maintains a persistent collection of non-dominated nodes during the whole search algorithm, and exploits it to also detect dominance relations across DD compilations.

Preliminary experiments convinced us to pursue the second strategy because of its much stronger pruning capacities and relatively small – or even positive – impact on the memory consumption of the algorithm, as will be discussed in Section 6. This way of enforcing the dominance rules involves storing all non-dominated nodes found at any stage of the B&B algorithm. We propose to use a hash table denoted by $Fronts_j$ for each DP stage j. Each of these hash tables stores *key-value* pairs of the form $\langle \kappa, Front \rangle$ that associate each dominance key κ with a Pareto front denoted $Front$ containing the set of non-dominated nodes. The only addition to the DD compilation procedure given by Algorithm 1 is that each layer is filtered through the dominance checks before expanding each of its nodes. The *pruned* set collects the pruned nodes of the layer and is used to define L'_j at line 6, a clone of the j-th layer from which the pruned nodes have been removed. In the rest of the algorithm, the pruned layer L'_j is employed instead of L_j to prevent generating any outgoing transition from the pruned nodes.

Algorithm 3 describes the actual dominance detection procedure, which also takes care of updating the *Fronts*. It begins by sorting the nodes of layer in *reverse* lexicographic order of dominance utilities Ψ at line 1. This ensures that if there exist two exact nodes $u_1, u_2 \in L_j$ such that $u_1 \succ u_2$, then u_1 will be processed before u_2 since $\Psi(u_1) \geq \Psi(u_2)$. Then, the algorithm loops over all nodes of the layer and first determines whether a front already exists for the dominance key of the current node. If not, it is simply initialized at lines 18 and 20 as a front containing the utility of the current node only. Otherwise, the existing front is retrieved as *Front* at lines 5 and 6 and the dominance check with respect to this front is initiated. By comparing the utility of the current node against those of the non-dominated nodes found so far, the node is declared dominated or not. In the dominated case, it is added to the *pruned* set at line 15, and otherwise to the *Front* at line 17. Along the way, every entry that the current

Algorithm 3 Dominance-based filtering of layer L_j of a DD $\mathcal{B}$.

```
1: sort nodes u in L_j in reverse lexicographic order of Ψ(u)
2: for all u ∈ L_j do
3:    if u is relaxed then
4:       continue
5:    if Fronts_j.contains(κ(σ(u))) then
6:       Front ← Fronts_j.get(κ(σ(u)))
7:       dominated ← False
8:       for all Ψ' ∈ Front do
9:          if Ψ(u) ≤ Ψ' then                          // exit if Ψ(u) is dominated
10:            dominated ← True
11:            break
12:         if Ψ(u) ≥ Ψ' then                  // remove entries that Ψ(u) dominates
13:            Front ← Front \ {Ψ'}
14:      if dominated then
15:         pruned ← pruned ∪ {u}
16:      else                                    // add to front if non-dominated
17:         Front ← Front ∪ {Ψ(u)}
18:   else                                       // initialize if first with given key
19:      Front ← {Ψ(u)}
20:      Fronts_j.insert(⟨κ(σ(u)), Front⟩)
```

node dominates is removed from the *Front* with line 13 to keep its size as small as possible. Note that this process is performed both for restricted and relaxed DDs, meaning that both types of DD benefit from this filtering mechanism. In addition, the exploratory nature of restricted DDs can help quickly find strong non-dominated nodes, and thus generate a lot of pruning early in the search.

Example 4. Using this procedure, we can derive dominance relations between nodes belonging to the DDs shown on Figure 2. Let us assume that the $Fronts$ have already been filled with the utilities of the nodes reached by the exact DD of Figure 2(a). We thus have that $Fronts_3 = \{\langle \mathbf{0}, \{(5,6),(11,1)\}\rangle\}$. Now if we consider layer L_3 of the exact DD given by Figure 2(b), we can first compute the utility – given by $\Psi(u) = (v^*(u \mid \mathcal{B}), \sigma(u))$ – of each node: $\Psi(c_4) = (0,12), \Psi(c_5) = (6,7), \Psi(c_6) = (1,6)$ and $\Psi(c_7) = (7,1)$, and then order the nodes by reverse lexicographic order of those, which produces: $\langle c_7, c_5, c_6, c_4\rangle$.

- Node c_7 with $\Psi(c_7) = (7,1)$ is dominated by utility $(11,1)$ in the front and is thus added to the *pruned* set.
- Node c_5 with $\Psi(c_5) = (6,7)$ is not dominated by any utility in the front and is thus added to the front. Moreover, it dominates the utility $(5,6)$, which is therefore removed from the front. This gives: $Fronts_3 = \{\langle \mathbf{0}, \{(11,1),(6,7)\}\rangle\}$.
- Node c_6 with $\Psi(c_6) = (1,6)$ is dominated by the utility that was just added to the front and is inserted in the *pruned* set.
- Finally, node c_4 with $\Psi(c_4) = (0,12)$ is added to the front because it has the largest remaining capacity and is therefore non-dominated. The final front is given by $Fronts_3 = \{\langle \mathbf{0}, \{(11,1),(6,7),(0,12)\}\rangle\}$.

5 Synergy with Caching

In [9], a caching mechanism was proposed to mitigate the number of repeated expansions of DP states, which are reached by multiple approximate DDs during the search. It includes a bottom-up procedure that computes an *expansion threshold* for each exact state reached by a relaxed DD, and that exploits the pruning inequalities of each filtering technique involved. Nodes are discarded whenever their value is lower or equal to the threshold. To combine dominance rules with this caching and pruning rule, we specify how expansion thresholds are computed in case of dominance pruning. Given a relaxed DD $\overline{\mathcal{B}}$, an exact node $u \in \overline{\mathcal{B}}$ with $\sigma(u) \in \mathcal{S}_j$ and a utility $\Psi' \in Fronts_j[\kappa(u)]$ such that $\Psi' \geq \Psi(u)$, the *dominance pruning threshold* of u is defined by, with $\Psi' = (v') \cdot \psi'$:

$$\theta_p(u \mid \overline{\mathcal{B}}) = \begin{cases} v' - 1, & \text{if } \psi' = \psi(u), \\ v', & \text{otherwise.} \end{cases}$$

Indeed, if u is dominated by a utility with the same partial utility, nodes with the same DP state will always be pruned unless they have a value of v' or higher. On the other hand, if u is dominated by a utility with a better partial utility, then nodes with the same DP state will always be pruned unless their value strictly exceeds v'. As these thresholds are propagated bottom-up in the relaxed DDs, dominance rules will also strengthen expansion thresholds for states of earlier DP stages, and can thus also reinforce the cache-based pruning strategy.

6 Computational Experiments

In this section, we evaluate experimentally the impact of dominance rules within the DD-based solver DDO [15]. To this end, four DP formulations were implemented and applied to the associated benchmark instances. For all problems, 600 seconds were given to solve each instance to optimality on a single thread, with the techniques described in [9,14] enabled by default. We first give a high-level description of the DP models and dominance rules of each problem, and of the benchmark instances and settings used before discussing the results of the experiments. Whereas the given definition of the dominance utility assumes that *greater is better*, the opposite rule is applied for minimization problems.

6.1 Experimental Setting

TSPTW The *Traveling Salesman Problem with Time Windows* is a variant of the well-known *Traveling Salesman Problem* where the cities are replaced by a set of customers $N = \{0, \ldots, n-1\}$ that must each be visited during a given time window. The objective is to find a tour starting and ending at customer 0 – the *depot* – and that visits all customers during their time window in the shortest possible time. The DP model used in the experiments is the one presented in [13], which extends the model introduced in [17] for the TSP. However, the present

description omits state components that are not relevant for the dominance rule, but are useful to tighten the relaxation of the problem.

- **DP model:** the state representation contains a tuple (c, t, M), where $c \in N$ and t respectively represent the customer and time of the last visit made by the salesman. The set $M \subseteq N$ contains the customers that still must be visited. Starting from the root state $\hat{r} = (0, 0, N)$, the transitions then model the possible next visits of the salesman and the associated cost.
- **Dominance rule:** if two states represent the salesman at the same location and having visited the same set of customers, then the one arriving earlier is always preferred. This dominance rule can be expressed by specifying the following dominance key: $\kappa(s) = (s.c, s.M)$. Then, the utility of a state is given by the elapsed time: $\psi(s) = s.t$. We could also simply have $\psi(s) = \mathbf{0}$ since the elapsed time is also captured by the node value. However, the definition given is also valid for the *travel time* version of the TSPTW.

All configurations of the DD-based solver were tested on a classical set of benchmark instances introduced in the following papers [1,11,22,28,30]. A dynamic width was used, where the maximum width for layers at depth j is given by $n \times (j + 1) \times \alpha$ with n the number of variables in the instance.

ALP The *Aircraft Landing Problem* requires to schedule the landing of a set of aircraft $N = \{0, \ldots, n-1\}$ on a set of runways $R = \{0, \ldots, r-1\}$. The aircraft have an earliest and latest landing time. Moreover, the set of aircraft is partitioned in disjoint sets $A_0, \ldots, A_{c-1}$ corresponding to c aircraft classes. For each pair of aircraft classes, a minimum separation time between the landings is given. The goal is to find a feasible schedule for all the aircraft, which minimizes the total waiting time – the delay between the earliest landing times and scheduled landing times – while respecting the latest landing times. The DP model presented in [24] was implemented, with a slightly different dominance rule.

- **DP model:** states are pairs (Q, ROP), with Q a vector that gives the remaining number of aircraft 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. The root state $\hat{r} = ((|A_0|, \ldots, |A_{c-1}|), ((0, \perp), \ldots, (0, \perp)))$ corresponds to the total number of aircraft to schedule for each class and an empty runway occupation profile, and the transitions model the next possible landings for each aircraft class and runway.
- **Dominance rule:** for a fixed remaining number of aircraft to schedule for each class and a same aircraft class previously scheduled on each runway, it is always better to have an earlier previous landing time if it comes with a better or equal objective function. This is expressed by the following dominance key and dominance utility vector: $\kappa(s) = (s.Q, (s.ROP_0.c, \ldots, s.ROP_{r-1}.c))$ and $\psi(s) = (s.ROP_0.l, \ldots, s.ROP_{r-1}.l)$.

A set of 720 random instances was generated, with $n \in \{25, 50, 75, 100\}$ aircraft, $r \in \{1, 2, 3, 4\}$ runways, and $c = 4$ aircraft classes. The target landing times

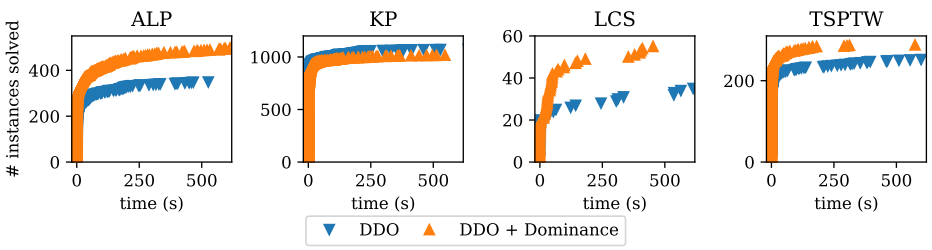


Fig. 3. Cumulative number of instances solved over time by DDO and DDO+D.

were generated by a Poisson arrival process with a mean inter-arrival time of $40/r$, instances with more runways thus require producing denser schedules. An arbitrary width of $W = 100$ is used for all experiments concerning the ALP.

LCS The *Longest Common Subsequence Problem* considers a set of m strings $S = \{S_0, \ldots, S_{m-1}\}$ and asks for the longest *subsequence* that appears in all of them. We reproduced the formulation presented in [18] almost identically.

- **DP model:** states are defined as tuples $\langle p_0, \ldots, p_{m-1} \rangle$ that give the current position in each string. The root state $\hat{r} = \langle 0, \ldots, 0 \rangle$ corresponds to the beginning of each string, and the transitions model the insertion of one character at the end of the subsequence and adapt the current positions.
- **Dominance rule:** Given states with the same current position in string S_0, it is always better to have both lower other positions and a greater objective value. This is expressed by the following dominance key and dominance utility vector: $\kappa(s) = \mathbf{0}$ and $\psi(s) = s$.

We used the following classical benchmark instances: BB [6], BL [5], Rat, Virus and Random [32], Poly and Abstract [27], but limited to instances with $m < 10$. A fixed width of 100 is used for all experiments concerning the LCS.

KP We solve the KP with all the ingredients presented in Examples 1 to 3. In addition, variables are ordered so that the items are considered in decreasing *profit-to-weight* ratios and the LP bound of [10] is used as an additional dual bound. A set of benchmark instances consisting of a random selection of 2% of the instances from [29] (636 instances) and 10% of the instances from [33] (530 instances). Again, a fixed width of 100 is used for all instances and configurations.

6.2 Results

Number of instances solved Figure 3 shows the number of instances solved by each solver and configuration with respect to the solving time. For all problems except the KP, DDO with dominance rules enabled – referred to as DDO+D

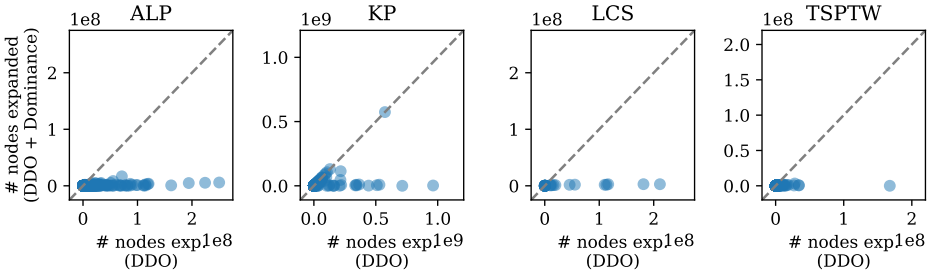


Fig. 4. Comparison of the number of node expansions performed by DDO and DDO+D for each instance solved by both configurations.

from now on – solves more instances than DDO, and by a large margin. As a result, we can confidently say that the integration of dominance rules has a very positive impact on the performance of the B&B algorithm. In the case of the KP, this low performance gain can be attributed to the fact that, when using the *profit-to-weight* ratio variable ordering and the LP bound, many unpromising partial solutions are quickly discarded and thus fewer dominance relations arise.

Number of node expansions The impact of the dominance rules can also be measured in terms of the number of nodes expanded during the successive approximate DD compilations. Figure 4 shows a pairwise comparison of this measure for each instance solved by both configurations. It appears that solving additional instances with DDO+D is made possible by a reduction in the number of node expansions needed to close them. Moreover, it clearly shows the magnitude of the filtering brought by the dominance rules, since for many instances that are unsolved by DDO, DDO+D requires only a negligible amount of node expansions to close them. For the KP, however, it seems that the decrease in node expansions is more significant than the decrease in time. This is probably due to the formulation of the dominance rule that needs to perform dominance checks for all pairs of nodes belonging to the same layer. For this kind of dominance checks, it might be worth considering a more specialized data structure than a simple list for the *Fronts*, such as k-d trees [2].

Quality of the first solution Another important dimension for a solver is the quality of the first solution found, which captures its *anytime* behavior. Figure 5 compares the value of the first solution found by DDO and DDO+D for each instance, as well as the iteration – in terms of B&B nodes – at which this solution is found for the TSPTW and the ALP. Indeed, those problems both have time window constraints that can make it difficult to find feasible solutions. This comparison allows us to make several observations for each problem:

– ALP: the quality of the first solution found by DDO+D is in general slightly better than the one obtained by DDO – in 314 cases, while the opposite

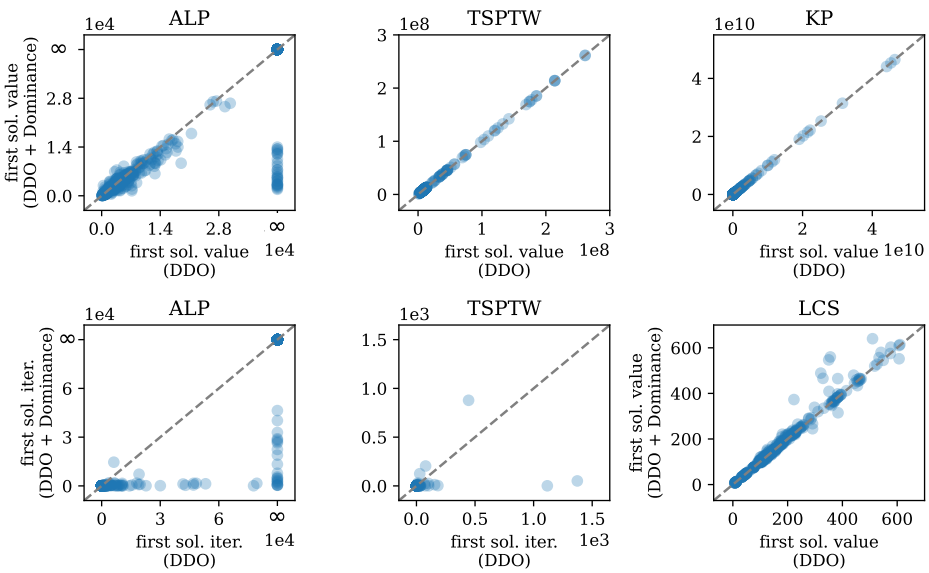


Fig. 5. Comparison of the value of the first solution obtained for each instance by DDO and DDO+D, and of the iteration at which it is found for ALP and TSPTW.

occurs in 118 cases. Furthermore, there are 29 instances for which DDO does not manage to find a single feasible solution to the problem, unlike DDO+D. In addition, when comparing the iteration at which the first solution is found, it appears that DDO+D finds a solution earlier than DDO for 235 instances, whereas the opposite is true for only 26 instances.

- TSPTW: the quality of the first solution found is only moderately impacted by the addition of dominance rules. Still, DDO+D finds a better first solution than DDO in 32 cases, against 4 cases in the other direction. Moreover, when looking at the iteration at which this first solution is found, we can see that DDO+D obtains it slightly earlier in the search for 34 instances, whereas the opposite is true for only 8 instances.
- LCS and KP: they are both maximization problems, so this time DDO+D compares better for data points located above the diagonal line. Although it is difficult to distinguish the solution values for the KP on Figure 5, DDO+D actually finds a slightly better solution than DDO in 229 cases, compared to only 20 cases in the opposite direction. For LCS, this occurs for 201 instances, whereas DDO finds a better first solution in 93 cases.

Integrating dominance rules therefore also contributes to quickly producing quality solutions by moving the compilation of restricted DDs away from parts of the search space that are not worth spending time on.

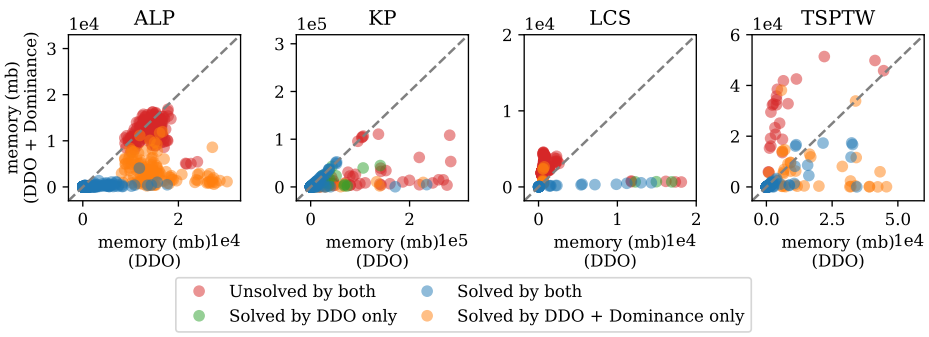


Fig. 6. Comparison of the peak amount of memory used for each instance by DDO and DDO+D, shown in different colors depending on which configurations solved it.

Memory consumption Finally, we discuss the memory footprint of maintaining the *Fronts* used to continually derive dominance relations with respect to non-dominated exact nodes previously found. Figure 6 compares the peak amount of memory used by both configurations for each instance. For all problems, we observe that a lower amount of memory is used by DDO+D for the large majority of instances solved by both configurations or only by DDO+D. Even for instances that both configurations fail to solve, DDO+D does not necessarily require more memory than DDO – the only exception is the TSPTW, although the maximum amount of memory used by DDO+D in those cases is not much larger than that reached by DDO for some instances.

7 Conclusion

In this paper, we proposed a formalism for specifying dominance rules of DP models. We then explained how they can be exploited within the DD compilation algorithm, as well as for the B&B algorithm as a whole by introducing a persistent data structure used to detect dominance relations across DD compilations. In addition, we showed how to combine this filtering mechanism with the caching procedure introduced in [9]. The modeling of the dominance rules was illustrated on four optimization problems and its impact was evaluated through extensive computational experiments. The results clearly highlight the interest of this additional ingredient, which significantly reduces the number of node expansions required by the algorithm to close the instances. This is directly reflected by the corresponding solving times, and leads to the resolution of many instances previously unsolved by DDO. Moreover, the experiments demonstrate the beneficial effect that dominance rules have in the ability of the algorithm to quickly find quality solutions, especially when the problem is highly constrained. Finally, the proposed memory-based dominance filtering procedure was shown to reduce the memory consumption of the solver in most cases.

References

1. Ascheuer, N.: Hamiltonian path problems in the on-line optimization of flexible manufacturing systems. Ph.D. thesis, University of Technology Berlin (1996)
2. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM **18**(9), 509–517 (1975)
3. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016)
4. Bianco, L., Mingozzi, A., Ricciardelli, S.: The traveling salesman problem with cumulative costs. Networks **23**(2), 81–91 (1993)
5. Blum, C., Festa, P.: Longest common subsequence problems. Metaheuristics for String Problems in Bioinformatics pp. 45–60 (2016)
6. Blum, C., Blesa, M.J.: Probabilistic beam search for the longest common subsequence problem. In: International Workshop on Engineering Stochastic Local Search Algorithms. pp. 150–161. Springer (2007)
7. Chambers, R.J., Carraway, R.L., Lowe, T.J., Morin, T.L.: Dominance and decomposition heuristics for single machine scheduling. Operations Research **39**(4), 639–647 (1991)
8. Chu, G., Stuckey, P.J.: Dominance breaking constraints. Constraints **20**, 155–182 (2015)
9. Coppé, V., Gillard, X., Schaus, P.: Decision diagram-based branch-and-bound with caching for dominance and suboptimality detection (2023)
10. Dantzig, G.B.: Discrete-variable extremum problems. Operations Research **5**(2), 266–277 (1957)
11. Dumas, Y., Desrosiers, J., Gelinas, E., Solomon, M.M.: An optimal algorithm for the traveling salesman problem with time windows. Operations research **43**(2), 367–371 (1995)
12. Fischetti, M., Salvagnin, D.: Pruning moves. INFORMS Journal on Computing **22**(1), 108–119 (2010)
13. Gillard, X.: Discrete optimization with decision diagrams: design of a generic solver, improved bounding techniques, and discovery of good feasible solutions with large neighborhood search. Ph.D. thesis, UCL-Université Catholique de Louvain (2022)
14. Gillard, X., Coppé, V., Schaus, P., Cire, A.A.: Improving the filtering of branch-and-bound mdd solver. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 231–247. Springer (2021)
15. Gillard, X., Schaus, P., Coppé, V.: 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. pp. 5243–5245 (2021)
16. Haahr, J.T., Pisinger, D., Sabbaghian, M.: A dynamic programming approach for optimizing train speed profiles with speed restrictions and passage points. Transportation Research Part B: Methodological **99**, 167–182 (2017)
17. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied mathematics **10**(1), 196–210 (1962)
18. Horn, M., Raidl, G.R.: A*-based compilation of relaxed decision diagrams for the longest common subsequence problem. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 72–88. Springer (2021)

19. Ibaraki, T.: The power of dominance relations in branch-and-bound algorithms. Journal of the ACM (JACM) **24**(2), 264–279 (1977)
20. Kohler, W.H., Steiglitz, K.: Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. Journal of the ACM (JACM) **21**(1), 140–156 (1974)
21. Kuroiwa, R., Beck, J.C.: Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 33, pp. 236–244 (2023)
22. Langevin, A., Desrochers, M., Desrosiers, J., Gélinas, S., Soumis, F.: A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. Networks **23**(7), 631–640 (1993)
23. Lee, J.H., Zhong, A.Z.: Exploiting functional constraints in automatic dominance breaking for constraint optimization. Journal of Artificial Intelligence Research **78**, 1–35 (2023)
24. Lieder, A., Briskorn, D., Stolletz, R.: A dynamic programming approach for the aircraft landing problem with aircraft classes. European Journal of Operational Research **243**(1), 61–69 (2015)
25. Mears, C., De La Banda, M.G.: Towards automatic dominance breaking for constraint optimization problems. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
26. Morrison, D.R., Jacobson, S.H., Sauppe, J.J., Sewell, E.C.: Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization **19**, 79–102 (2016)
27. Nikolic, B., Kartelj, A., Djukanovic, M., Grbic, M., Blum, C., Raidl, G.: Solving the longest common subsequence problem concerning non-uniform distributions of letters in input strings. Mathematics **9**(13), 1515 (2021)
28. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transportation Science **32**(1), 12–29 (1998)
29. Pisinger, D.: Where are the hard knapsack problems? Computers & Operations Research **32**(9), 2271–2284 (2005)
30. Potvin, J.Y., Bengio, S.: The vehicle routing problem with time windows part ii: genetic search. INFORMS Journal on Computing **8**(2), 165–172 (1996)
31. Righini, G., Salani, M.: Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. Computers & Operations Research **36**(4), 1191–1203 (2009)
32. Shyu, S.J., Tsai, C.Y.: Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Computers & Operations Research **36**(1), 73–91 (2009)
33. Smith-Miles, K., Christiansen, J., Muñoz, M.A.: Revisiting where are the hard knapsack problems? via instance space analysis. Computers & Operations Research **128**, 105184 (2021)