Advances in Discrete Optimization with Decision Diagrams: Dominance, Caching and Aggregation-Based Heuristics

Vianney Coppé

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

January 2024

ICTEAM
Louvain School of Engineering
UCLouvain
Louvain-la-Neuve
Belgium

Advances in Discrete Optimization with Decision Diagrams: Dominance, Caching and Aggregation-Based Heuristics
by Vianney Coppé

Abstract

The branch-and-bound algorithm based on decision diagrams introduced by Bergman et al. in 2016 is a framework for solving discrete optimization problems with a dynamic programming formulation. It works by compiling a series of bounded-width decision diagrams that can provide lower and upper bounds for any given subproblem. Eventually, every part of the search space will be either explored or pruned by the algorithm, thus proving optimality. This thesis presents new ingredients to speed up this search algorithm. First, it describes how dominance rules can be utilized during the compilation of decision diagrams to detect and filter dominated nodes. The second contribution is a caching mechanism that fully exploits the structure of dynamic programming models and the way their state space is explored by the branch-and-bound algorithm. Its key idea is to prevent the repeated expansion of nodes corresponding to the same dynamic programming state by querying expansion thresholds cached throughout the search. These thresholds are based on dominance relations between partial solutions previously found and on the pruning inequalities of the filtering techniques introduced by Gillard et al. in 2021. Finally, a procedure for deriving dual bounds and node selection heuristics through aggregate dynamic programming is detailed. Assuming a maximization problem, relaxed decision diagrams provide upper bounds through state merging while restricted decision diagrams obtain lower 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. This can be better captured by the proposed bounds and heuristics because they are acquired by pre-solving a so-called aggregate version of the problem. Extensive computational experiments show that the pruning brought by these additional filtering techniques and heuristics allows reducing the number of nodes expanded by the algorithm significantly and finding quality solutions earlier in the search. This results in more benchmark instances of difficult optimization problems being solved in less time, and in tighter optimality gaps when instances cannot be solved under the given time limit.

Acknowledgments

If the doctoral journey is often perceived as a very solitary experience, its achievement is undoubtedly the fruit of multiple ingredients which cannot come together without a community around it, namely: support, encouragement, advice, exchange of ideas and knowledge. With these few words, I would like to thank all the people who constituted this community for me, and contributed in one way or another to the success of my journey.

First, I would like to thank Pierre, my advisor, who sparked my interest in research before I even considered pursuing a Ph.D., and managed to get me on board with this adventure. Furthermore, his support and guidance were instrumental in making me believe in my work when it seemed futile, and in channeling my ideas when they were going in too many directions.

My second thank you goes to my colleague and collaborator Xavier, who laid a very solid foundation for easily experimenting with decision diagrams, always knew how to challenge my ideas, and helped me on many occasions when I felt lost with his favorite programming language.

Next, I would like to thank all my colleagues in the INGI department for fostering a friendly environment. In particular, those with whom I shared an office – Auguste, Benoît, Gaël, Hélène and Nicolas – always in a relaxed mood. Moreover, those among Achille, Alexander, Alexandre, Auguste, Augustin, Benoît, Charles, Damien, Donatien, François, Guillaume, Harold, Hélène, Magali, Mathias, Nicolas, Lucile and Xavier with whom I had good times whether playing spikeball, music, card games, doing crosswords, sharing drinks, or enjoying local culture during conferences. I would like to extend my gratitude to the administrative and technical staff of the department who facilitated countless procedures and continuously maintained a reliable infrastructure for our computational experiments.

My thanks also go to all the wonderful housemates with whom I have had the chance to live over the last four years, and to all my other close friends who gravitate around. Thank you for the fun, sporty and caring atmosphere that surely helped me maintain my balance.

Finally, I would like to thank my parents, my brothers and their beautifully growing families, for their everlasting support and love, which brought me to where I am now. This of course also applies to Eléonore, whose presence by my side brightens my days and whose natural curiosity has always been interested in my work.

Contents

Introduction | 1

1.1 Context

Combinatorial optimization plays a crucial role in addressing some of the most complex challenges in the world across various domains such as logistics, transportation, manufacturing, healthcare and energy production and distribution. It provides a set of efficient and reliable tools to assist and facilitate decision-making processes when facing problems that are too complex or cumbersome for us humans to solve. Several optimization paradigms have been widely adopted to tackle real-world problems, namely *Mixed-Integer Programming* (MIP), *Constraint Programming* (CP) and *Boolean satisfiability* (SAT). *Dynamic programming* (DP) is another popular technique to model and solve difficult combinatorial problems. Although this resolution technique is appealing, the process of memorizing all solved subproblems rapidly becomes infeasible in terms of memory, as the required memory size can grow exponentially with the input size, rendering the computation intractable. Discrete optimization with *decision diagrams* (DDs) is a recent framework for addressing the memory issue for solving constraint optimization problems using their DP formulation. Apart from offering new modeling perspectives, this technique can exploit the compactness of DP models within an adapted *branch-and-bound* (B&B) algorithm introduced by [Ber+16]. In addition to conducting the search within the DP state space, the strength of DD-based B&B lies in its dedicated approach to deriving lower and upper bounds. Both are obtained by compiling bounded-width DDs, respectively called *restricted* and *relaxed* DDs – assuming a maximization problem. Those approximate DDs each have their own strategy for limiting their size, which works either by removing or merging excess nodes.

The DDs used today in the field of discrete optimization originate from compact encodings of Boolean functions, first known as binary decision programs [Lee59] and later as *binary decision diagrams* (BDDs) [Ake78; Bry86]. *Multi-valued decision diagrams* (MDDs) were then suggested by [KB90] as an extension of BDDs to variables and functions taking values from discrete sets. These different variants of DDs were successfully applied in different domains such as formal verification [Hu95], model checking [CGL94], computer-aided design [Min95] and optimization [LPV94; HS97; Bec+05; HH06; HH07]. More recently, [And+07] introduced relaxed DDs that act as a constraint store for constraint programming solvers. These DDs can represent a superset of the feasible variable assignments and their size can be controlled by bounding their width to balance computational cost and the filtering strength. Relaxed DDs were then adapted by [Ber+14a] as a mean of deriving upper bounds for discrete optimization problems that can be modeled with DP [Hoo13]. They were followed closely by their lower bounding counterparts called restricted DDs [Ber+14b], which represent a subset of the solution space.

Several improvements to the B&B algorithm based on these two ingredients have been suggested since its introduction: [Gil+21] proposed additional bounding procedures named *local bounds* and *rough upper bounds* to enhance the pruning of the B&B and thus speed up the search. When the relaxed DDs are compiled by separation in the B&B algorithm, [RCR22; RCR23] explained how intermediate relaxed DDs could be stored in the B&B queue to avoid compiling them from scratch at each iteration. The bounds provided from approximate DDs can also be improved by discovering variable orderings that yield more compact DDs. [Cap+22] designed a reinforcement learning approach to perform this task while [KH22] proposed different portfolio mechanisms to dynamically select the best ordering among a predefined set of alternatives. Finally, another promising research direction is the integration of DD-based optimization components to CP solvers, as driven by the HADDOCK modeling language and constraint store [GMH20; GMH22; GMH23]. For a complete overview of the latest theoretical contributions and applications of DDs in the field of discrete optimization as a whole, we refer the reader to the survey by [CCB22].

1.2 Research Goals

In this thesis, we aim to further enhance the capabilities of the DD-based discrete optimization paradigm by investigating additional filtering mechanisms. By generating extra pruning during the top-down compilation of approximate DDs throughout the search, the B&B algorithm needs to actually explore a smaller part of the search space in order to find and prove the optimal solution. In addition, the DDs generated are sparser and therefore require less computation time to obtain. Furthermore, restricted DDs have a better chance of finding good solutions quickly, and the bounds derived from relaxed DDs are tighter, since those approximate DDs cover a smaller portion of the search space. Because better primal and dual bounds only intensify the pruning power of the B&B algorithm, this creates a virtuous circle, which we wish to further strengthen.

Our quest for supplementary filtering procedures revolved around the following research questions:

- Can DD-based optimization benefit from simple dominance rules?

- Is the amount of DP states that are visited by multiple approximate DDs during the search significant? Can we reduce it? How does it impact the B&B algorithm?

- Can aggregate dynamic programming be used to derive information that would efficiently guide and tighten the approximate DDs, compared to existing heuristics and relaxation schemes?

1.3 Contributions

The main contributions of this thesis include several ingredients that participate in advancing the field of discrete optimization with decision diagrams, following the line of research described in the previous section, and the application of these to a wide range of combinatorial optimization problems, namely:

- a formalism for defining dominance rules arising in DP models and a procedure for systematically exploiting them within the DD compilation algorithm and the B&B as a whole.

- a problem-agnostic caching mechanism that takes full advantage of the information discovered inside the successive relaxed DDs compiled to derive and store expansion thresholds that condition the future expansion of nodes associated with already reached DP states.

- a node selection heuristic for restricted DDs and a complementary relaxation scheme that leverage *aggregate dynamic programming* to better capture the general structure of a problem, and a formalism for specifying an aggregation for each given problem.

- the extension of a DP model for the *Single-Row Facility Layout Problem* that handles several types of additional constraints very efficiently, as well as a *rough lower bound* for this problem.

- the implementation of all the above contributions inside the open source DD-based optimization solver *DDO* [GSC21], and the specification of multiple problem formulations via its modeling interface. This allowed performing extensive experiments for various optimization problems and configurations of the solver with the same code base.

1.4 Publications

Most of the contributions presented in this thesis have been developed in separate research papers, mentioned here in the order in which they will be discussed:

- V. Coppé, X. Gillard, and P. Schaus. "Decision Diagram-Based Branch-and-Bound with Caching for Dominance and Suboptimality Detection". In: (2023). arXiv: `2211.13118`. This paper introduces a specialized caching mechanism for the DD-based B&B algorithm. It is currently under review.

- V. Coppé, X. Gillard, and P. Schaus. "Boosting Decision Diagram-Based Branch-And-Bound by Pre-Solving with Aggregate Dynamic Programming". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023. It explains how the aggregate dynamic programming principle can be applied to the DD-based optimization framework, in the form of node selection heuristics and complementary relaxation scheme.

- V. Coppé, X. Gillard, and P. Schaus. "Solving the Constrained Single-Row Facility Layout Problem with Decision Diagrams". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022. This paper presents a MIP and a DD-based approach for solving the *Constrained Single-Row Facility Layout Problem.* It also describes a *breadth-first* B&B algorithm and explains its benefits over the classical *best-first* variant.

This thesis also led to the following publication: V. Coppé and P. Schaus. "A Conflict Avoidance Table for Continuous Conflict-Based Search". In: *Proceedings of the International Symposium on Combinatorial Search.* Vol. 15. 1. 2022, pp. 264–266, in which we generalize a key component of the best-performing solvers for the Multi-Agent Path Finding problem in discrete time and space to the more realistic case of continuous time and space. It will not be discussed here as it falls outside of otherwise coherent research efforts about discrete optimization with decision diagrams.

1.5 Outline

The thesis begins with a general introduction in Chapter 2 about DDs and how, given a DP model, they can encode solutions to a discrete optimization problem. It culminates with a detailed description of the DD-based B&B algorithm introduced by [Ber+16], and of the improvements presented in [Gil+21]. Chapter 3 then describes how dominance rules can be consistently formulated for DP models, and how the DD compilation algorithm can be adapted to efficiently detect dominated nodes and filter them. Computational results covering four optimization problems and demonstrating the importance of dominance rules conclude the chapter. Another filtering mechanism is introduced in Chapter 4, based on the caching of an expansion threshold for DP states reached by exact nodes of relaxed DDs. This expansion threshold provides a condition on the minimum path length required so that new paths reaching previously visited DP states have a chance to improve on the best solution currently found. This ingredient is evaluated on the four problems studied in Chapter 3 as well as two additional problems, and is shown to significantly decrease the number of node expansions needed for the B&B algorithm to terminate the search, resulting in considerable speedups. In Chapter 5, a framework for incorporating aggregate dynamic programming into the DD-based B&B is proposed. It explains how node selection heuristics and dual bounds can be obtained by solving an aggregate – i.e. relaxed and easier — version of the problem. The framework is applied to three of the optimization problems tackled in the previous chapters, and its impact in terms of solution quality and additional pruning is discussed. Throughout the first chapters mentioned, all the concepts introduced are illustrated with a single *Bounded Knapsack Problem* instance. Finally, Chapter 6 presents two approaches for solving a constrained facility layout problem with MIP and DDs, and compares their effectiveness in handling different types of constraints. Chapter 7 then steps back and reflects on the contributions of this thesis, as well as on some open questions and perspectives for future investigations.

Preliminaries | 2

This chapter provides an overview of the DD-based discrete optimization framework. It starts with a reminder on how discrete optimization problems can be modeled with *dynamic programming* (DP). Next, the exact and inexact representation of a DP formulation by DDs is explained. In particular, *restricted* and *relaxed* DDs are defined as well as the top-down compilation algorithm used to obtain them. The chapter continues with a description of the DD-based B&B algorithm exploiting the lower and upper bounds derived from these approximate DDs. Finally, the more advanced pruning techniques introduced in [Gil+21] are recalled. A toy example of the Bounded Knapsack Problem is developed throughout the chapter to illustrate how the modeling components are formulated and provide a visual depiction of the algorithms.

2.1 Discrete Optimization

A discrete optimization problem $\mathcal{P}$ involves finding the best possible solution x^* from a finite set of feasible solutions $Sol(\mathcal{P}) = \mathcal{D} \cap C$. This set is determined by the domain $\mathcal{D} = \mathcal{D}_0 \times \cdots \times \mathcal{D}_{n-1}$ from which the variables $x = (x_0, \ldots, x_{n-1})$ each take on a value, i.e. $x_j \in \mathcal{D}_j$, and by a set C modeling all the constraints that solutions need to satisfy. The quality of the solutions is evaluated according to an objective function $f(x)$ that must be maximized. Formally, the problem is defined as $\max \{f(x) \mid x \in \mathcal{D} \cap C\}$ and any optimal solution x^* must satisfy $x^* \in Sol(\mathcal{P})$ and $\forall x \in Sol(\mathcal{P}) : f(x^*) \geq f(x)$.

Example 2.1.1. *Given a set of items $N = \{0, \ldots, n-1\}$ with weights $W = \langle w_0, \ldots, w_{n-1} \rangle$, values $V = \langle v_0, \ldots, v_{n-1} \rangle$ and quantities $Q = \langle q_0, \ldots, q_{n-1} \rangle$, the goal of the Bounded Knapsack Problem (BKP) is to choose the number of copies of each item to include in the knapsack so that the total value is maximized, and the total weight is kept under a given capacity C. A classical MIP formulation of this problem would rely on a vector of integer variables $x = (x_0, \ldots, x_{n-1})$ where x_j corresponds to the decision of including x_j copies of item j in the knapsack. Therefore, the domain of each variable x_j is $\mathcal{D}_j = \left\{0, \ldots, q_j\right\}$. Furthermore, the only constraint of the problem is to respect the capacity of the knapsack, so $C = \left\{x \mid \sum_{j=0}^{n-1} x_j w_j \leq C\right\}$. Finally, the objective function is the total value of the selected items, given by $f(x) = \sum_{j=0}^{n-1} x_j v_j$.*

2.2 Dynamic Programming

DP is a *divide-and-conquer* strategy introduced by Bellman [Bel54] for solving discrete optimization problems with an inherent recursive structure. It works by recursively decomposing the problem into 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:

- a vector of *control variables* $x = (x_0, \dots, x_{n-1})$ with domain $\mathcal{D} = \mathcal{D}_0 \times \cdots \times \mathcal{D}_{n-1}$ such that $x_j \in \mathcal{D}_j$ with $j \in \{0, \dots, n-1\}$.
- a *state space* $\mathcal{S}$ partitioned into $n+1$ sets $\mathcal{S}_0, \dots, \mathcal{S}_n$ that correspond to distinct stages of the DP model. In particular, $\mathcal{S}_j$ contains all states having j variables assigned. Several special states are also defined: the *root* $\hat{r}$, the *terminal* $\hat{t}$ and the *infeasible* state $\hat{0}$.
- a set t of *transition functions* s.t. $t_j : \mathcal{S}_j \times \mathcal{D}_j \longrightarrow \mathcal{S}_{j+1}$ 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 $d \in \mathcal{D}_j$.
- a set h of *transition value functions* s.t. $h_j : \mathcal{S}_j \times \mathcal{D}_j \longrightarrow \mathbb{R}$ representing the immediate reward of assigning some value $d \in \mathcal{D}_j$ to the variable x_j for $j = 0, \dots, n-1$.
- a *root value* v_r to account for constant terms in the objective.

If those elements can be defined for $\mathcal{P}$, then finding the optimal solution is equivalent to solving:

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

Example 2.2.1. *The BKP has a well-known DP formulation that we recall here.*

- *Like in the MIP formulation of the problem, it uses one variable $x_j \in \left\{0, \dots, q_j\right\}$ for each item $j \in N$ that decides the number of copies of it to include in the knapsack.*

- *The states simply contain the remaining capacity of the knapsack. Indeed, when taking the decisions sequentially, the exact same sets of items can be added to all partial solutions with the same remaining capacity, independently of the items selected before. The state space is thus defined as $S = [0, C]$, with the root state $\hat{r} = C$ starting at maximum capacity.*

- *The transition functions are given by:*

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

  It means that the weight of item j is subtracted from the remaining capacity as many times as the number of copies selected. If the transition violates the capacity constraint, the transition is redirected to the infeasible state $\hat{0}$.

- *The transition value functions $h_j(s^j, x_j) = x_j v_j$ simply add the value of item j for each copy included in the knapsack.*

- *The root value is $v_r = 0$.*

2.3 Decision Diagrams

In the context of discrete optimization, a DD is a graphical representation of the set of solutions to a given problem $\mathcal{P}$. DDs are well suited to encode and manipulate compact formulations such as DP models, due to their ability to preserve the uniqueness of overlapping subproblems. In mathematical terms, a decision diagram $\mathcal{B} = (U, A, \sigma, l, v)$ is a layered directed acyclic graph consisting of a set of nodes U that are connected by a set of arcs A. Each node is mapped to a DP state by the function σ. The set of nodes U can be partitioned into *layers* $L_0, \ldots, L_n$ corresponding to the successive stages of the DP model, each containing one node for each *distinct* state of the corresponding stage. Therefore, arcs $a = (u_j \xrightarrow{d} u_{j+1})$ connect nodes of consecutive layers $u_j \in L_j, u_{j+1} \in L_{j+1}$ and represent the transition between states $\sigma(u_j)$ and $\sigma(u_{j+1})$. The *label* $l(a) = d$ of an arc encodes the decision that assigns the value $d \in \mathcal{D}_j$ to variable x_j. The value $v(a)$ of the arc stores 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. Each $r \rightsquigarrow t$ path $p = (a_0, \ldots, a_{n-1})$ that traverses the DD from top to bottom through arcs $a_0, \ldots, a_{n-1}$ represents a solution $x(p) = (l(a_0), \ldots, l(a_{n-1}))$ to $\mathcal{P}$. Its objective value is given by the accumulation of the arc values along the path and the root value: $v(p) = v_r + \sum_{j=0}^{n-1} v(a_j)$. Finally, the set of all solutions appearing in the DD is defined as $Sol(\mathcal{B}) = \{x(p) \mid \exists p : r \rightsquigarrow t, p \in \mathcal{B}\}$ and $\mathcal{B}$ is said

V	W	Q
2	4	1
3	6	1
6	4	2
6	2	2
1	5	1
$C = 15$		

Table 2.1: A BKP instance.

exact if it perfectly represents the set of solutions of the corresponding problem, i.e. $Sol(\mathcal{B}) = Sol(\mathcal{P})$ and $v(p) = f(x(p)), \forall p \in \mathcal{B}$. For convenience, all nodes u and paths p appearing in a DD $\mathcal{B}$ can be accessed respectively with the notation $u \in \mathcal{B}$ and $p \in \mathcal{B}$.

Example 2.3.1. *Let us consider the BKP instance given by Table 2.1 with 5 items and a knapsack capacity of 15. The exact DD obtained by developing the DP model introduced in Example 2.2.1 for this instance is represented on Figure 2.1. The bold path is the longest path of the DD and corresponds to the optimal solution* $x^* = (0, 0, 2, 2, 0)$ *with value* $f(x^*) = 24$.

Before detailing the DD compilation algorithm, let us provide some additional elements of notation. We denote by $p^*(u_1 \rightsquigarrow u_2 \mid \mathcal{B})$ an optimal path between nodes u_1 and u_2 in a DD $\mathcal{B}$ and by $v^*(u_1 \rightsquigarrow u_2 \mid \mathcal{B})$ its value. For a DD $\mathcal{B}$ rooted at a node u_r, we assume that an exact $r \rightsquigarrow u_r$ path $p(u_r) = p^*(r \rightsquigarrow u_r \mid \mathcal{B}')$ was found and attached to u_r during the prior compilation of another DD $\mathcal{B}'$. For conciseness, we denote by $p^*(u \mid \mathcal{B}) = p(u_r) \cdot p^*(u_r \rightsquigarrow u \mid \mathcal{B})$ an optimal path connecting the root node r of the problem and node u within the DD $\mathcal{B}$, with $\cdot$ the concatenation operator. The value of this path is given by $v^*(u \mid \mathcal{B}) = v(p(u_r) \cdot p^*(u_r \rightsquigarrow u \mid \mathcal{B})) = v(p(u_r)) + v^*(u_r \rightsquigarrow u \mid \mathcal{B})$. Furthermore, we write $p^*(\mathcal{B}) = p^*(t \mid \mathcal{B}), v^*(\mathcal{B}) = v^*(t \mid \mathcal{B})$ and $x^*(\mathcal{B}) = x(p^*(\mathcal{B}))$ to refer to one of the longest $r \rightsquigarrow t$ paths in $\mathcal{B}$, its value and the corresponding variable assignment. Finally, we define the *successors* of a node $u \in \mathcal{B}$ as the set of nodes reachable from u in $\mathcal{B}$, including u itself: $Succ(u \mid \mathcal{B}) = \{u\} \cup \{u' \mid (u \rightsquigarrow u') \in \mathcal{B}\}$.

2.3.1 Compilation

Given the subproblem $\mathcal{P}|_{u_r}$ that restricts $\mathcal{P}$ to paths traversing some node u_r, Algorithm 1 details the top-down compilation of a DD rooted at u_r, given that $\sigma(u_r)$ is a state at the i-th stage of the DP model. The notations utilized in the algorithms are deliberately less formal to make the explanations easier to understand.

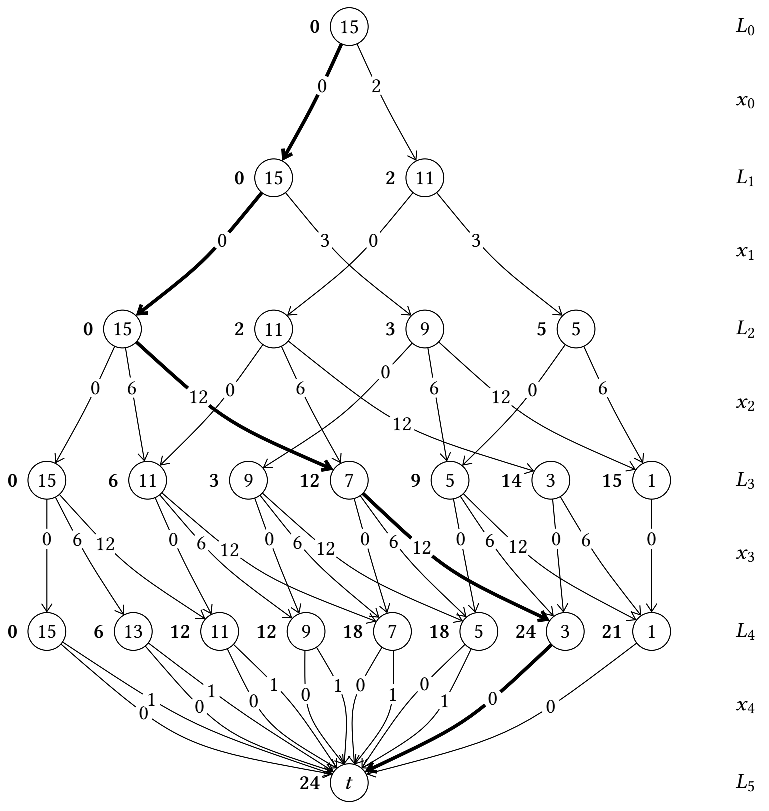


Figure 2.1: The exact DD for the BKP instance of Table 2.1. The value inside each node corresponds to its state – the remaining capacity – and the annotation on the left gives the value of the longest path that reaches it. For clarity, only arc values are present. The longest path is highlighted in bold.

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 7 to 13, until all variables are assigned. Once the DD is fully unrolled, line 14 merges all nodes of the terminal layer into a single terminal node t. At line 4, the *width* of the current layer is computed and compared against a parameter W called the *maximum width.* When the layer width exceeds W, the algorithm yields approximate DDs by either restricting or relaxing the layer at line 5. In order to obtain an exact DD, one can simply set $W = \infty$. 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*

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

```
1: $i \longleftarrow$ index of the layer containing $u_r$
2: $L_i \longleftarrow \{u_r\}$
3: for $j = i$ to $n - 1$ do
4:     if $|L_j| > W$ then
5:         restrict or relax the layer to get $W$ nodes with Algorithm 2
6:     $L_{j+1} \longleftarrow \emptyset$
7:     for all $u \in L_j$ do
8:         if $v^*(u \mid \mathcal{B}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}$ then        // RUB pruning
9:             continue
10:        for all $d \in \mathcal{D}_j$ do
11:            create node $u'$ with state $\sigma(u') = t_j(\sigma(u), d)$
               or retrieve it from $L_{j+1}$
12:            create arc $a = (u \xrightarrow{d} u')$ with $v(a) = h_j(\sigma(u), d)$ and $l(a) = d$
13:            add $u'$ to $L_{j+1}$ and add $a$ to $A$
14: merge nodes in $L_n$ into terminal node $t$
```

DDs. These two variants of DDs are both capable of maintaining the width of all layers under a given maximum width but rely on different strategies to reduce the width of a layer. The former one yields lower bounds and feasible solutions while the latter can be used to derive upper bounds.

A note about reduced DDs A DD is *reduced* [Bry86; Weg00] if it does not contain any *equivalent* subgraphs, that is, isomorphic subgraphs where corresponding nodes belong to the same layer and corresponding arcs share the same labels. DD reduction is very important for many applications, since it produces a unique DD of minimal size for a given variable ordering. However, *weighted* DDs used for optimization are less amenable to reduction since arc values further restrict which arcs can be superimposed. [Hoo13] explained how transition cost functions of DP models can be rearranged to allow compiling reduced weighted DDs. Yet, this transformation is not always possible nor necessary since DDs are already reduced in some sense, indeed, they are derived from DP models that superimpose equivalent subproblems by design.

2.3.2 Restricted DDs

The compilation of a restricted DD can be seen as a *beam search* in the DP state space. At line 5 of Algorithm 1, restriction is thus synonymous with removing surplus nodes. As detailed by Algorithm 2, the least promising nodes of the layer are selected according to a heuristic and simply dropped before resuming the compilation as normal. As a result, the compilation of a restricted DD

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 $L'_j$
3:     $L_j \leftarrow L_j \setminus \mathcal{M}$
4:     create node $\mu$ with state $\sigma(\mu) = \oplus(\sigma(\mathcal{M}))$        // for relaxation only
       and add it to $L_j$
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 $v(a') = \Gamma_{\mathcal{M}}(v(a), u)$
```

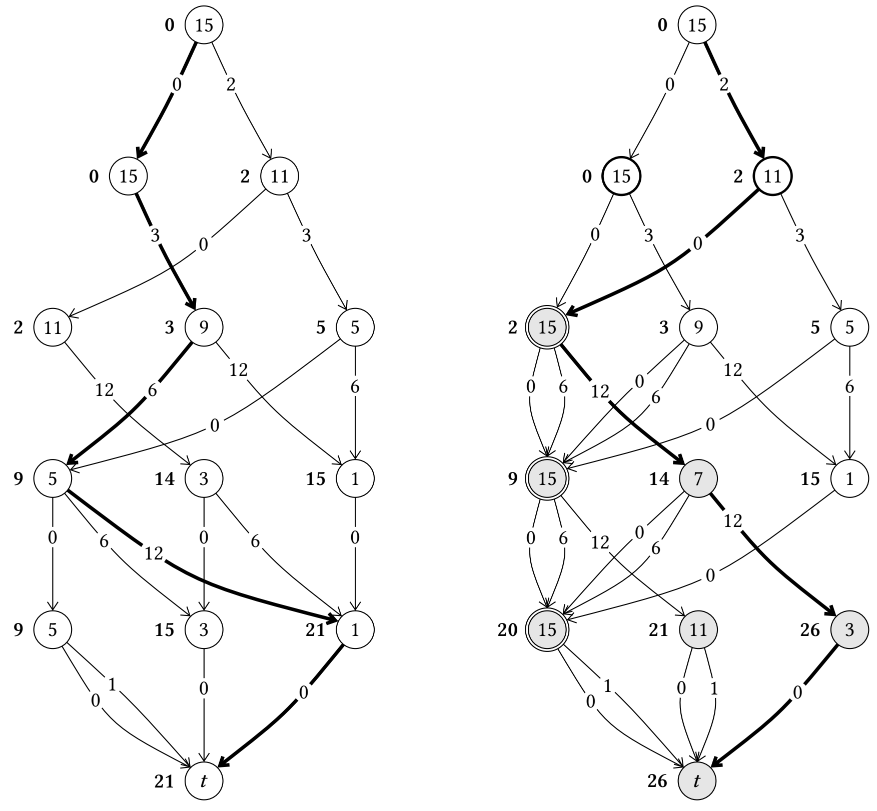
will produce a subset of the solutions of the problem, those remaining solutions being feasible since the transitions were left untouched. For a restricted DD $\underline{\mathcal{B}}$, we thus have $Sol(\underline{\mathcal{B}}) \subseteq Sol(\mathcal{P})$ and $v(p) = f(x(p)), \forall p \in \underline{\mathcal{B}}$.

Example 2.3.2. *Figure 2.2(a) shows the result of compiling a restricted DD for the BKP instance of Table 2.1 with a maximum width of 3. By applying the greedy heuristic that deletes nodes with the lowest prefix values, the best solution that the restricted DD thus compiled contains is* $\underline{x} = x^*(\underline{\mathcal{B}}) = (0, 1, 1, 2, 0)$, *which gives a lower bound of* $\underline{v} = v^*(\underline{\mathcal{B}}) = 21$.

2.3.3 Relaxed DDs

As opposed to restricted DDs that encode a subset of the solutions, the purpose of relaxed DDs is to represent all solutions of the problem. This is achieved by locally relaxing the problem by *merging* nodes together. By doing so, no feasible solutions will be removed, but infeasible ones might be introduced. It requires defining problem-specific merging operators to merge the corresponding DP states. 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. The resulting state should encompass all merged states and preserve all their outgoing transitions. 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 $\Gamma_{\mathcal{M}}$ can be specified to adjust the value of the arcs incident to the merged node at line 6. As the merging operator gives a relaxed representation of all merged states, it can introduce infeasible outgoing transitions. In addition, merging nodes u_1, u_2 allows combining any $r \rightsquigarrow u_1$ path with any $u_2 \rightsquigarrow t$ path and vice-versa. Given a valid relaxation operator, we can write for any relaxed DD $\overline{\mathcal{B}}$ that $Sol(\overline{\mathcal{B}}) \supseteq Sol(\mathcal{P})$ and $v(p) \geq f(x(p)), \forall p \in \overline{\mathcal{B}}$. To correctly explore the search space, we distinguish *exact nodes* from *relaxed nodes.* A node u in a DD $\mathcal{B}$ rooted at node u_r is said *exact* if, for any $r \rightsquigarrow u_r \rightsquigarrow u$ path in $\mathcal{B}$, applying all transitions specified



(a) Restricted DD (b) Relaxed DD

Figure 2.2: (a) A restricted and (b) a relaxed DD for the BKP instance of Table 2.1(a), as compiled with Algorithm 1 with $W = 3$. (b) Relaxed nodes are colored in gray and merged nodes are circled twice.

by its arcs recursively from the root leads to the state $\sigma(u)$. All nodes that do not meet this criterion are called *relaxed.*

Example 2.3.3. *As explained in Example 2.2.1, the states of the DP model for the BKP are identified by the remaining capacity of the knapsack. In order to create a relaxed representation of the states to merge, it suffices to keep the maximum remaining capacity among them. This is formally written as:* $\oplus(\sigma(\mathcal{M})) = \max_{s \in \sigma(\mathcal{M})} s$. *For this simple formulation, the operator* $\Gamma_{\mathcal{M}}$ *can be defined as the identity function since there is no need to modify the arc values.*

Given these relaxation operators, one can obtain the relaxed DD of width 3 that is represented on Figure 2.2(b). The longest path in this diagram corresponds to the solution $\overline{x} = x^*(\overline{\mathcal{B}}) = (1, 0, 2, 2, 0)$ *for a value of 26 and a total weight of 16. This solution thus violates the capacity constraint, which was allowed to happen because the only effect of the merging operators given above is precisely to relax* *this constraint. Nevertheless, the value of this infeasible solution provides an upper bound $\overline{v} = v^*(\overline{\mathcal{B}}) = 26$ for the problem.*

2.3.4 Rough Upper Bound Pruning

Before presenting how approximate DDs are embedded in a B&B scheme, this section covers the last modeling component that affects the compilation algorithm. The *rough upper bound* (RUB) introduced by [Gil+21] aims at identifying suboptimal nodes as early as during the compilation of approximate DDs. It does so by computing a cheap problem-specific upper bound $\overline{v}_{rub}(\sigma(u))$ for every node at line 8 of Algorithm 1, which can help pruning many nodes before even generating their successors by comparing it with a known lower bound $\underline{v}$ obtained from an initial solution. The RUB can be specified for the states of a given DP model, such that $\overline{v}_{rub}(\sigma(u)) \geq v^*(u \rightsquigarrow t \mid \mathcal{B})$ for any node u in $\mathcal{B}$ the exact DD for problem $\mathcal{P}$. Since the RUB is computed for each node of the approximate DDs, it needs to be computationally cheap. However, a tight RUB 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.

Example 2.3.4. *A naive RUB for the BKP can simply add the maximum quantity of all remaining items to the knapsack, disregarding the capacity constraint. This can be written as $\overline{v}_{rub}(s^j) = \sum_{k=j}^{n-1} q_k v_k$. Of course, a better choice could be to use the LP bound introduced by [Dan57].*

Figure 2.3 shows a relaxed DD for the BKP instance of Table 2.1 compiled from the root with $W = 3$, using RUB and LocB pruning this time. Given the lower bound of $\underline{v} = 21$ obtained in Example 2.3.2, RUB pruning can successfully discard nodes c_1, c_2, c_3, d_1 and d_2. For instance, the RUB for all nodes of the fourth layer is given by $\overline{v}_{rub}(\sigma(u)) = q_3 v_3 + q_4 v_4 = 2 \times 6 + 1 \times 1 = 13$. This pruning improves the quality of the relaxed DD as only one layer resorted to node merging, compared to three for the relaxed DD of Figure 2.2(b).

2.4 Branch-and-Bound

The ingredients presented in Section 2.3 allow building a DD-only B&B algorithm, as introduced in [Ber+16]. Restricted DDs are used to quickly find quality feasible solutions from any starting node while relaxed DDs recursively decompose the problem at hand and provide bounds for the subproblems thus generated. The heart of the algorithm lies in the concept of *exact cutset* that dictates how to exhaustively explore the entire search space.

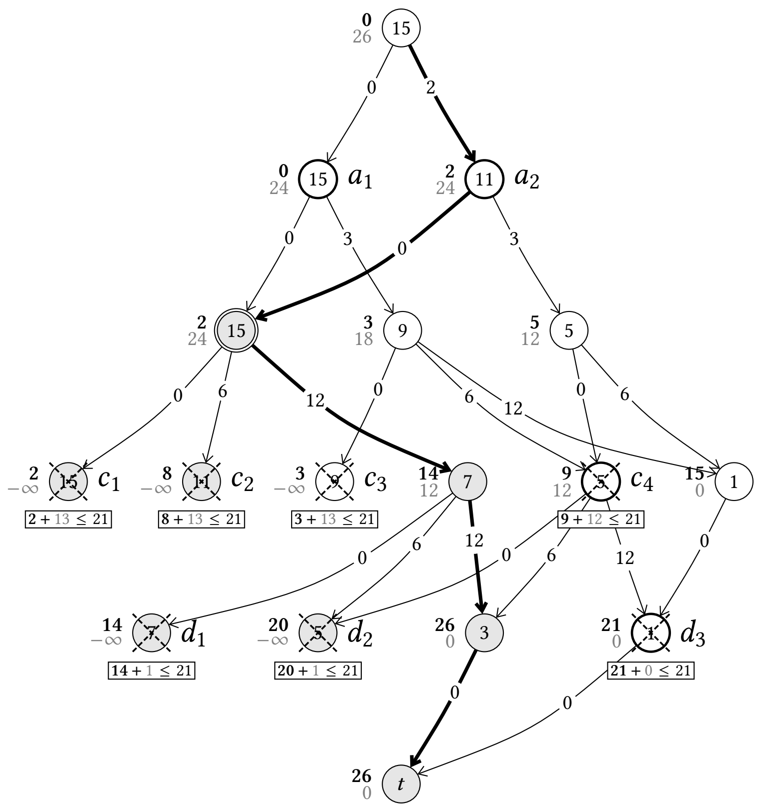


Figure 2.3: A relaxed DD for the BKP instance of Table 2.1(a), as compiled with Algorithm 1 with $W = 3$ and RUB and LocB pruning enabled. The LocBs are annotated in gray on the left of each node. Pruning decisions are detailed below the filtered nodes.

2.4.1 Exact Cutsets

As explained in Section 2.3.3, relaxed DDs offer a way to derive an upper bound for any given subproblem from which the compilation is initiated. They can, however, serve a second purpose when aiming to solve problems to optimality. Considering a relaxed DD $\overline{\mathcal{B}}$ rooted at a node u_r, the divide-and-conquer nature of the underlying DP model suggests that the corresponding subproblem $\mathcal{P}|_{u_r}$ can be decomposed in a set of other subproblems given by its direct successors in $\overline{\mathcal{B}}$. In other words, solving all these subproblems separately is equivalent to solving the subproblem in u_r. By transitivity, an exact decomposition of u_r can be formed by exact nodes belonging to layers further down the diagram. This decomposition principle is formalized by the notion of *exact cutset* (EC) of $\overline{\mathcal{B}}$.

Definition 2.4.1 (Exact cutset)**.** *An exact cutset $EC(\overline{\mathcal{B}})$ of a relaxed DD $\overline{\mathcal{B}}$*

rooted at node u_r is a set of exact nodes such that any $u_r \rightsquigarrow t$ path crosses at least one of them.

As a result, we can state that solving $\mathcal{P}|_{u_r}$ is equivalent to solving $\mathcal{P}|_u$ for all $u \in EC(\overline{\mathcal{B}})$. In the B&B algorithm, ECs can thus be used to identify which subproblems must still be processed as to enumerate – or prune – all possible solutions. However, there can exist many sets of exact nodes conforming to the above definition for a given relaxed DD $\overline{\mathcal{B}}$. In [Ber+16], three different strategies to identify an EC are described.

- *Traditional branching*: if u_r belongs to layer L_i, *traditional branching* (TB) decomposes u_r into the subproblems obtained by assigning all possible values to variable x_i, and thus consists of all the nodes in the following layer $TB(\overline{\mathcal{B}}) = L_{i+1}$.
- *Last exact layer*: the *last exact layer* (LEL) is the deepest layer in $\overline{\mathcal{B}}$ that contains only exact nodes.

$$LEL(\overline{\mathcal{B}}) = L_{j'} \text{ where } j' = \operatorname*{arg\,max}_{j=i,\dots,n} \forall u \in L_j : u \text{ is exact}$$

- *Frontier cutset*: the *frontier cutset* (FC) contains all exact nodes at the frontier between exact and relaxed nodes in $\overline{\mathcal{B}}$.

$$FC(\overline{\mathcal{B}}) = \left\{ u \in \overline{\mathcal{B}} \mid u \text{ is exact } \wedge \exists a = (u \xrightarrow{d} u') \text{ such that } u' \text{ is relaxed} \right\}$$

Example 2.4.1. *Using both TB and LEL, the EC of the relaxed DD of Figure 2.3 would contain nodes a_1 and a_2. The FC would consist of node a_1, a_2, c_4 and d_3 since they all have one relaxed direct successor.*

2.4.2 Local Bounds

Equally important as the EC are the upper bounds prescribing whether a subproblem is worth exploring. In the original DD-based B&B algorithm presented in [Ber+14a], the only criterion for filtering a node retrieved from the EC of a relaxed DD $\overline{\mathcal{B}}$ was the comparison between single upper bound given by $v^*(\overline{\mathcal{B}})$ and the best known lower bound $\underline{v}$. The *local bounds* (LocBs) presented by [Gil+21] refine this reasoning by computing a distinct upper bound for each node u in $\overline{\mathcal{B}}$, given by the value of the longest $u \rightsquigarrow t$ path in $\overline{\mathcal{B}}$.

Definition 2.4.2 (Local bound)**.** *Given $\overline{\mathcal{B}}$ a relaxed DD for problem $\mathcal{P}$ and a node $u \in \overline{\mathcal{B}}$, the local bound $\overline{v}_{locb}(u \mid \overline{\mathcal{B}})$ of u within $\overline{\mathcal{B}}$ is given by:*

$$\overline{v}_{locb}(u \mid \overline{\mathcal{B}}) = \begin{cases} v^*(u \rightsquigarrow t \mid \overline{\mathcal{B}}), & \textit{if } (u \rightsquigarrow t) \in \overline{\mathcal{B}}, \\ -\infty, & \textit{otherwise.} \end{cases}$$

Algorithm 3 Computation of the local bound of every node u in $\overline{\mathcal{B}}$.

```
1:  $i \longleftarrow$ index of the root layer of $\overline{\mathcal{B}}$
2:  $(L_i, \ldots, L_n) \longleftarrow Layers(\overline{\mathcal{B}})$
3:  $\overline{v}_{locb}(u \mid \overline{\mathcal{B}}) \longleftarrow -\infty$
4:  $mark(u) \longleftarrow$ false **for each node** $u \in \overline{\mathcal{B}}$
5:  $\overline{v}_{locb}(t \mid \overline{\mathcal{B}}) \longleftarrow 0$
6:  $mark(t) \longleftarrow$ true
7:  **for** $j = n$ **down to** $i$ **do**
8:      **for all** $u \in L_j$ **do**
9:          **if** $mark(u)$ **then**
10:             **for all** arc $a = (u' \longrightarrow u)$ incident to $u$ **do**
11:                 $\overline{v}_{locb}(u' \mid \overline{\mathcal{B}}) \longleftarrow \max\left\{\overline{v}_{locb}(u' \mid \overline{\mathcal{B}}), \overline{v}_{locb}(u \mid \overline{\mathcal{B}}) + v(a)\right\}$
12:                 $mark(u') \longleftarrow$ true
```

Example 2.4.2. *In Figure 2.3, below the value of the longest path shown next to each node, we add the LocB in gray. Given the incumbent solution computed in Figure 2.2(a) and supposing an FC is used, the LocBs allow to deduce that* c_4 *and* d_3 *are not worth exploring further. Indeed, for each of them we have* $v^*(u \mid \overline{\mathcal{B}}) + \overline{v}_{locb}(u \mid \overline{\mathcal{B}}) = 21 \leq \underline{v} = 21$.

LocBs can be computed efficiently by performing a bottom-up traversal of $\overline{\mathcal{B}}$, as formalized by Algorithm 3. In the algorithm, the *mark* flags are propagated to all nodes that have at least one path reaching the terminal node t. In parallel, the value $v^*(u \rightsquigarrow t \mid \overline{\mathcal{B}})$ of each marked node u is computed by accumulating the arc values traversed upwards. At the terminal node, $\overline{v}_{locb}(t \mid \overline{\mathcal{B}})$ is set to zero at line 5. For all other nodes, the value $\overline{v}_{locb}$ is updated by their direct successors at line 11.

To simplify the exposition of the B&B algorithm and of the concepts introduced in Chapter 4, we define the upper bound $\overline{v}(u \mid \overline{\mathcal{B}})$ that combines the RUB and the LocB obtained after compiling a relaxed DD $\overline{\mathcal{B}}$ and with respect to a known lower bound $\underline{v}$.

$$\overline{v}(u \mid \overline{\mathcal{B}}) = \begin{cases} \overline{v}_{rub}(\sigma(u)), & \text{if } v^*(u \mid \overline{\mathcal{B}}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}, \\ \min\left\{\overline{v}_{rub}(\sigma(u)), \overline{v}_{locb}(u \mid \overline{\mathcal{B}})\right\}, & \text{otherwise.} \end{cases}$$

In the case where the RUB caused the node to be pruned during the compilation at line 8 of Algorithm 1, the value of the RUB is kept. Otherwise, the minimum of both upper bounds is retained to maximize the pruning potential.

Algorithm 4 The DD-based branch-and-bound algorithm.

```
1:  $Fringe \leftarrow \{r\}$   // a priority queue ordered by decreasing $v(u) + \overline{v}(u)$
2:  $\underline{x} \leftarrow \perp, \underline{v} \leftarrow -\infty$   // incumbent solution and its value
3:  while $Fringe$ is not empty do
4:      $u \leftarrow$ best node from $Fringe$, remove it from $Fringe$
5:      if $v(u) + \overline{v}(u) \leq \underline{v}$ then
6:          continue
7:      $\underline{\mathcal{B}} \leftarrow Restricted(u)$   // compile restricted DD with Algorithm 1
8:      if $v^*(\underline{\mathcal{B}}) > \underline{v}$ then   // update incumbent
9:          $\underline{x} \leftarrow x^*(\underline{\mathcal{B}}), \underline{v} \leftarrow v^*(\underline{\mathcal{B}})$
10:     if $\underline{\mathcal{B}}$ is not exact then
11:         $\overline{\mathcal{B}} \leftarrow Relaxed(u)$   // compile relaxed DD with Algorithm 1
12:         compute LocBs with Algorithm 3 applied to $\overline{\mathcal{B}}$
13:         for all $u' \in EC(\overline{\mathcal{B}})$ do
14:             $v(u') \leftarrow v^*(u' \mid \overline{\mathcal{B}}), \overline{v}(u') \leftarrow \overline{v}(u' \mid \overline{\mathcal{B}}), p(u') \leftarrow p^*(u' \mid \overline{\mathcal{B}})$
15:             if $v(u') + \overline{v}(u') > \underline{v}$ then
16:                 add $u'$ to $Fringe$
17: return $(\underline{x}, \underline{v})$
```

2.4.3 Algorithm

The B&B algorithm is formalized by Algorithm 4 and illustrated by Figure 2.4. It relies on a priority queue – referred to as the *Fringe* – that maintains the set of open nodes throughout the search. The *Fringe* is initialized with the root node r corresponding to the root state of the DP model $\sigma(r) = \hat{r}$. Then, line 3 loops over the nodes contained in the *Fringe* and for each of them, a restricted DD $\underline{\mathcal{B}}$ is compiled at line 7. It may provide a solution with value $v^*(\underline{\mathcal{B}})$ improving the current best, in which case the incumbent is updated at lines 8 to 9. If $\underline{\mathcal{B}}$ is exact, meaning that no nodes were removed from the DD, then this subproblem can be considered as fully explored. Otherwise, a relaxed DD $\overline{\mathcal{B}}$ must be developed at line 11 as to decompose the node into smaller subproblems given by the EC of $\overline{\mathcal{B}}$. Thanks to the RUB and LocB computations, an upper bound $\overline{v}(u)$ is derived for each node $u \in EC(\overline{\mathcal{B}})$. It is then used to discard nodes that are guaranteed to lead to solutions of value worse or equal to $\underline{v}$. This check is performed at line 15 before adding a node to the *Fringe* and is repeated at line 5 when a node is selected for exploration. In addition to the upper bound $\overline{v}(u)$, line 14 also attaches to each cutset node u the longest $r \rightsquigarrow u$ path $p(u) = p^*(u \mid \overline{\mathcal{B}})$ and its value $v(u) = v^*(u \mid \overline{\mathcal{B}})$. The cutset nodes having passed the pruning test are then added to the *Fringe* in order to pursue the exhaustive exploration of the search space, and this whole process is repeated until the *Fringe* is emptied.

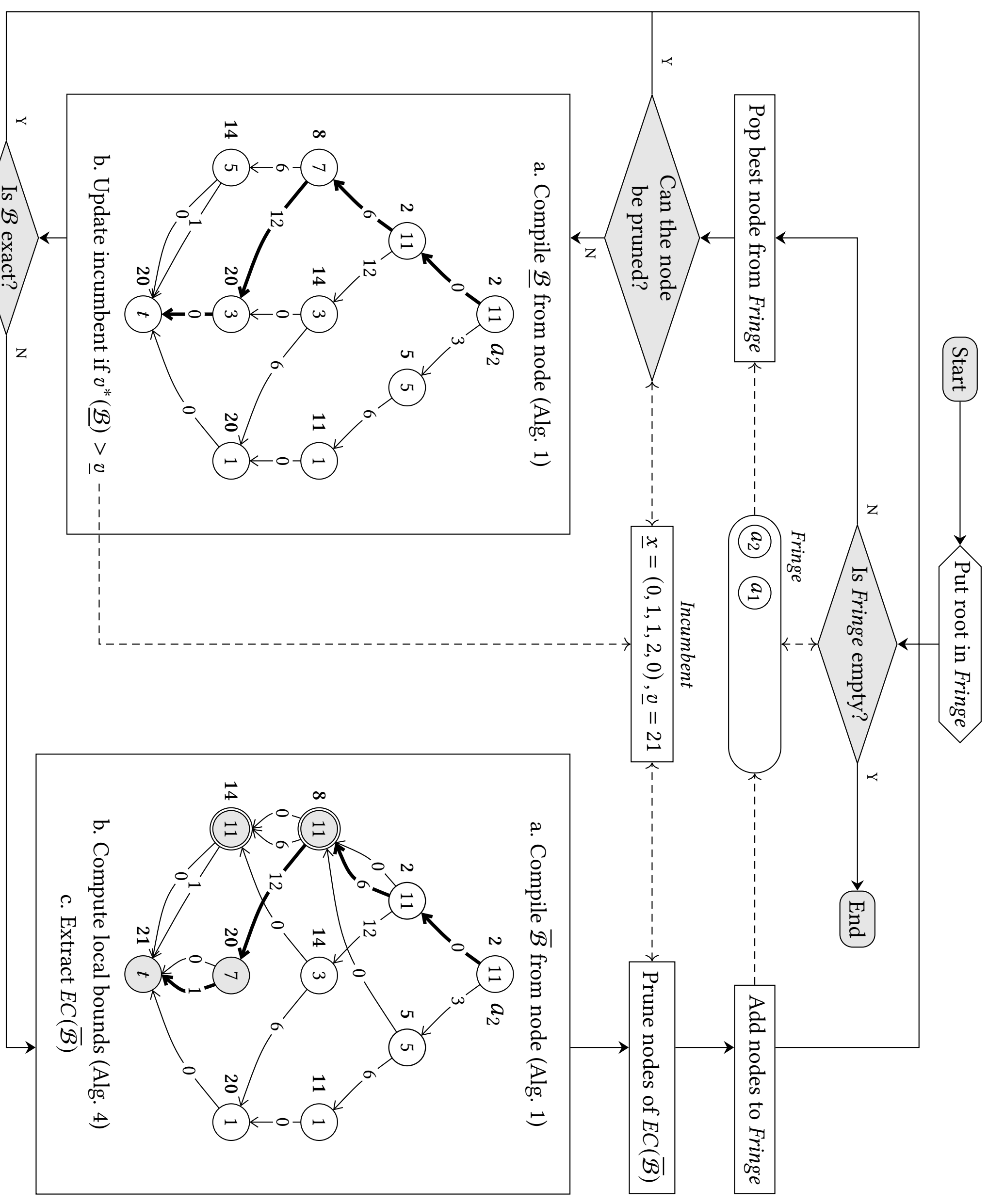

Start
Put root in *Fringe*
Is *Fringe* empty?
Y
N
End
Pop best node from *Fringe*
Fringe
a_1
a_2
Add nodes to *Fringe*
Can the node be pruned?
Y
N
Incumbent
$\underline{x} = (0,1,1,2,0)$, $\underline{v} = 21$
Prune nodes of $EC(\overline{\mathcal{B}})$
a. Compile $\underline{\mathcal{B}}$ from node (Alg. 1)
b. Update incumbent if $v^*(\underline{\mathcal{B}}) > \underline{v}$
Is $\underline{\mathcal{B}}$ exact?
Y
N
a. Compile $\overline{\mathcal{B}}$ from node (Alg. 1)
b. Compute local bounds (Alg. 4)
c. Extract $EC(\overline{\mathcal{B}})$

Figure 2.4: Flowchart of the DD-based B&B algorithm. It depicts the state of the algorithm after nodes a_1 and a_2 have been added to the *Fringe* from the FC of the relaxed DD of Figure 2.3 and node a_2 is selected for exploration.

Example 2.4.3. *Figure 2.4 illustrates the execution of the main loop of the B&B algorithm after having added the FC of the relaxed DD of Figure 2.3 to the Fringe. As mentioned in Example 2.4.1, this FC consists of nodes a_1, a_2, c_4 and d_3 but recall from Example 2.4.2 that c_4 and d_3 can be pruned thanks to their LocB and the lower bound derived from Figure 2.2(a). Among the Fringe, node a_2 has the best upper bound and is selected for exploration. Indeed, $v(a_2) + \overline{v}(a_2) = 2 + 24 = 26$ whereas $v(a_1) + \overline{v}(a_1) = 0 + 24 = 24$. Since a_2 cannot be pruned, the left part of the flowchart shows the compilation of a restricted DD from this node, which fails to find a solution with a better value than 21. As the restricted DD is not exact, a relaxed DD is developed and shown in the right part of the flowchart. The upper bound given by this relaxed DD is $v^*(\overline{\mathcal{B}}) = 21$, which is equal to the incumbent value. It is thus unnecessary to enqueue any node from the EC.*

2.5 Variants and Heuristics for Compilation

Section 2.4 concluded the overview of the DD-based B&B framework introduced in [Ber+16] and of the extensions proposed in [Gil+21]. This section provides more details about some components of the algorithm, and about the variants that exist in the literature.

2.5.1 Node Selection Heuristic

In the DD compilation algorithm, a heuristic decision must be made about which nodes to remove or merge when a layer exceeds the maximum width allowed – this occurs at line 2 of Algorithm 2. The most commonly used strategy is the *MinLP* heuristic that selects nodes with the minimum longest path value. It was introduced in [Ber+14a] and shown to yield much tighter bounds than a *random* strategy as well as a problem-specific heuristic for the *Maximum Independent Set Problem* (MISP).

While the *MinLP* strategy is well suited for the compilation of restricted DDs, it may be ineffective in the case of relaxed DDs. Indeed, it is completely oblivious to the similarity, or rather the dissimilarity, between the states that will be merged. To address this issue, [FR20] suggested node merging procedures incorporating state similarity measures and longest path values, and that were shown to perform well on the MISP and the *Set Cover Problem.*

Another strategy for increasing the similarity of nodes merged together was proposed in [Hor+21; HR21], using a problem-specific node *labeling function* that conditions which nodes can be merged together. Different labeling strategies were shown to produce very different results in terms of bound quality, DD size and compilation time for a prize-collecting scheduling problem and for the *Multiple Longest Common Subsequence Problem* (MLCS).

2.5.2 Variable Ordering

So far, we assumed that DDs are compiled by considering the variables in their natural ordering $x_0, \ldots, x_{n-1}$. For some DP models such as those for sequencing problems [Hoo17; Hoo19], this ordering is imposed by the way the recursion is formulated. However, for some other problems, nothing prevents from tweaking the variable ordering so that decisions about crucial elements of a problem are made first. We distinguish two types of variable orderings:

- a *static* variable ordering is a permutation of the ordering of the variables that remains fixed throughout the whole algorithm,
- a *dynamic* variable ordering dynamically selects the next variable to assign when developing a new layer in the DD compilation algorithm. This means that the approximate DDs compiled during a given execution do not necessarily consider the variables in the same order.

[Ber+12; Ber+14a] presented several static and dynamic variable orderings for the MISP and showed that they greatly impact the size of the DDs compiled and the quality of their bounds. Another promising research direction is to automatically discover good variable orderings for a given class of problems. Recently, [Cap+22] designed a reinforcement learning approach to perform this task while [KH22] proposed different portfolio mechanisms to dynamically select the best ordering among a predefined set of alternatives.

Example 2.5.1. *For the BKP, considering the variables associated to the items with the best value-to-weight ratio first yields much better results.*

2.5.3 Long Arcs

Another simplification made in this chapter is that transitions concerning a variable x_j are applied to all nodes of the previous layer L_j indistinctively, creating an arc and a successor node for each valid variable assignment. *Long arcs* allow modeling transitions that skip some variables having no impact on the state associated with a given node [Ber+14a] For problems that allow it, long arcs reduce the size of the DDs since they represent multiple transitions. However, as discussed in detail in [Gil22], bounding the number of nodes generated when compiling DDs with long arcs is not straightforward. Indeed, while each layer can be reduced to a maximum of W nodes, the size of the set of nodes that are skipping a given is unbounded. [Gil22] discusses this issue in detail and provides the adapted DD compilation algorithm that supports long arcs.

Example 2.5.2. *When developing a layer by applying transitions concerning variable x_j of the BKP, nodes for which the remaining capacity is insufficient to* *insert a single copy of item j can be put aside until reaching the next item that fits, i.e. variable x_j is skipped for node u if $\sigma(u) < w_j$.*

2.5.4 Alternative Compilation Schemes

In Section 2.3, we recalled the *top-down* DD compilation procedure used in the original DD-based B&B paper [Ber+16], and reproduced here by Algorithm 1. Although this procedure is the only one capable of producing both restricted, relaxed and exact DDs, there exist other techniques for compiling relaxed or exact DDs. Namely, the so-called compilation by *incremental refinement* [Had+08; HHH10] relies on node *separation* instead of node *merging*. It starts with a dummy relaxed DD of width one that contains all the solutions to the problem and potentially also many infeasible solutions. The algorithm then iteratively refines each layer by filtering infeasible outgoing arcs and by splitting nodes associated with relaxed states, until all possible splits are performed, or the maximum width is reached. Incremental refinement proves particularly valuable in scenarios where new constraints or assignments must be incorporated iteratively, because it suffices to initiate a new refinement pass to include those new requirements into the DD.

- For instance, [And+07] suggested relaxed DDs acting as a variable domain store for CP solvers. By applying propagation algorithms for multiple constraints on a single DD, their filtering strength can be combined. The HADDOCK language and system [GMH20; GMH22; GMH23] provides a unified and extensible framework for specifying and employing DD-based constraints within CP solvers.

- More recently, *column elimination* [Hoe20; KH23] was proposed as an alternative to *column generation*. Whereas column generation operates on a restricted set of variables – corresponding to *columns* – that is iteratively extended, column elimination uses a relaxed set of columns, compactly represented by a relaxed DD. This set of columns is iteratively reduced by adding new constraints to the relaxed DD as long as inconsistencies can be found in the solution. Very promising results were presented for the *Graph Coloring Problem* and the *Capacitated Vehicle Routing Problem*.

- Incremental refinement can also be used to compile relaxed DDs within DD-based B&B solvers. Moreover, [RCR22; RCR23] recognized that if relaxed DDs are stored in the *Fringe* instead of nodes, then the compilation of a subsequent relaxed DD can be warm-started by extracting the subgraph rooted at a chosen cutset node from a previously compiled relaxed DD – an operation referred to as *peeling*.

Closer to the top-down compilation procedure, [Hor+21; HR21] proposed an *A^* compilation* scheme that expands nodes in a best-first fashion. By proceeding in this way, relaxed DDs are not developed layer by layer, and one cannot simply limit the width of each layer. Instead, the size of the A*-like open list is bounded, and an adapted merging mechanism that allows merging nodes across layers is used when the open list grows too large. This alternative compilation scheme is shown to produce tighter bounds than the other techniques mentioned so far. [Hor+21] also explained how the information contained in a previously obtained relaxed DD can help steer the compilation of a restricted DD.

Additionally, a *local search* framework for compiling relaxed DDs was presented in [RCR18], combining ingredients from top-down compilation and incremental refinement. Through sequences of elementary manipulations that respect the maximum width of the DD, the approach iteratively tightens the bound by splitting nodes along the current longest path.

Finally, [GS22] added new ingredients to the top-down compilation algorithm to perform *large neighborhood search* with restricted DDs. It allows generating new solutions in the neighborhood of the best known solution by preserving the corresponding path during the compilation, while introducing some randomization when deciding which nodes to remove from a layer exceeding the maximum width.

2.6 The *DDO* Library

DDO [GSC21] is an *open source*[1], *generic* and *efficient* implementation of the DD-based B&B algorithm written in Rust. It was and is still developed by Xavier Gillard, who described the architecture of the library and its implementation details in great depth in [Gil22]. It provides several simple modeling interfaces that allow formulating new problems easily, and solving them without needing to re-implement or actually fully understand the whole solver. Furthermore, it abstracts many of the heuristic decisions that arise in the B&B and DD compilation algorithms. Therefore, different configurations can be created and tested smoothly without encroaching on the solver code. Finally, DDO was implemented with concurrency in mind and allows to distribute the computational load to multiple threads without any additional effort needed from the user. The theoretical contributions presented in the coming chapters of this thesis have all been implemented and evaluated inside DDO.

[1]Available at `https://github.com/xgillard/ddo`.

Dominance Rules | 3

This chapter presents the integration of dominance rules to the DD compilation algorithm, and to the DD-based B&B algorithm as a whole. This is currently unpublished work, yet this ingredient is essential to solve certain problems efficiently, as the reader will be able to realize.

3.1 Introduction

As shown in [Gil+21] with the RUB, 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 helps to obtain deeper ECs, as illustrated by Example 2.3.4. This results in fewer cutset nodes being enqueued in the *Fringe*, which means the algorithm can solve the same problem by exploring much fewer nodes overall.

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 [KS74; Iba77] in the general case of a B&B framework. Several optimization paradigms successfully applied them, including MIP [FS10], CP [CS15; MD15; LZ23] and DP [Cha+91; BMR93; RS09; HPS17]. 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 chapter is, to the best of our knowledge, the first to fill this gap for this particular field of research. It starts by providing general definitions about dominance rules within the context of DD-based optimization in Section 3.2, and describe how dominance rules can be formulated for DP models. Section 3.3 then explains how they can be exploited to systematically detect and prune dominated nodes during the search. The BKP running example of Chapter 2 is pursued to illustrate the proposed modeling components and the associated filtering procedure. Finally, we present in Sections 3.4 and 3.5 the modeling and the experimental evaluation of the integration of dominance rules for the *0–1 Knapsack Problem* (KP), the *Traveling Salesman Problem with*

Time Windows (TSPTW), the *Aircraft Landing Problem* (ALP) and the *Multiple Longest Common Subsequence* problem (MLCS).

3.2 Definitions and Modeling Ingredients

Let us first formally define the concept of node dominance in the DD-based optimization context. Note that the following definitions only concern exact nodes. Indeed, because relaxed nodes give an approximate representation of multiple nodes, they would not produce valid dominance relations.

Definition 3.2.1 (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 $\kappa : \mathcal{S} \longrightarrow \mathcal{S}'$ 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 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 $\psi : \mathcal{S} \longrightarrow \mathbb{R}^k$ 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.

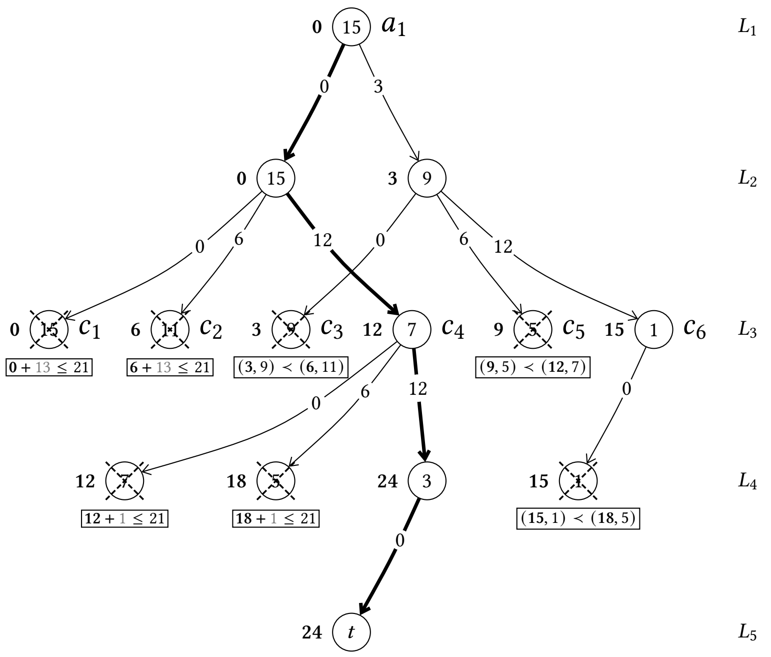


Figure 3.1: An exact DD compiled from node a_1 of Figure 2.3 and exploiting the dominance rule for this problem. The dominance relation is given below each node pruned successfully.

The following definition formalizes the connection between those operators and Definition 3.2.1, 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 3.2.2 (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 3.2.1. *In the case of the BKP, a node u_1 having both higher value and higher remaining capacity than another node u_2 will always produce better solutions. This dominance rule can be formulated through the following dominance operators:*

- *The dominance key of a state s is a zero-dimensional vector $\kappa(s) = \mathbf{0}$ since there are no other restrictions than comparing states of the same DP stage.*
- *The partial dominance utility is the identity function $\psi(s) = s$ so that for a node $u \in \mathcal{B}$, the dominance utility operator compares both 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 3.1 shows an exact DD compiled from node a_1 *of Figure 2.3, obtained by performing dominance checks using the rule defined above during the compilation. Three dominated nodes can be pruned, resulting in a DD of width 2 only when ignoring the pruned nodes. For instance, node* c_3 *is dominated by* c_2 *since* $\kappa(c_3) = \kappa(c_2) = \mathbf{0}$ *and* $\Psi(c_3) = (3, 9) \leq \Psi(c_2) = (6, 11)$.

3.3 Filtering the Search Using Dominance Rules

Now that we have described how dominance rules can be modeled and how a dominance relation between two nodes can be identified, we explain how to systematically detect and prune dominated nodes within the DD-based B&B algorithm. In this regard, we propose two strategies that both fall in the category of *memory-based dominance relations* [Mor+16].

- 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 3.5. 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 DD layer 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. These fronts are initialized at line 2 of the adapted B&B given by Algorithm 5, which is otherwise unchanged. The adapted DD compilation procedure given by Algorithm 6 is also mostly untouched, except 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 5 The DD-based branch-and-bound algorithm.

```
1: $Fringe \leftarrow \{r\}$        // a priority queue ordered by decreasing $v(u) + \overline{v}(u)$
2: $Fronts_j \leftarrow \emptyset$ for $j = 0, \dots, n$ // hash tables of dom. keys to Pareto fronts
3: $\underline{x} \leftarrow \bot, \underline{v} \leftarrow -\infty$        // incumbent solution and its value
4: while $Fringe$ is not empty do
5:     $u \leftarrow$ best node from $Fringe$, remove it from $Fringe$
6:     if $v(u) + \overline{v}(u) \leq \underline{v}$ then
7:         continue
8:     $\underline{\mathcal{B}} \leftarrow Restricted(u)$        // compile restricted DD with Algorithm 6
9:     if $v^*(\underline{\mathcal{B}}) > \underline{v}$ then        // update incumbent
10:        $\underline{x} \leftarrow x^*(\underline{\mathcal{B}}), \underline{v} \leftarrow v^*(\underline{\mathcal{B}})$
11:    if $\underline{\mathcal{B}}$ is not exact then
12:        $\overline{\mathcal{B}} \leftarrow Relaxed(u)$        // compile relaxed DD with Algorithm 6
13:        compute LocBs with Algorithm 3 applied to $\overline{\mathcal{B}}$
14:        for all $u' \in EC(\overline{\mathcal{B}})$ do
15:            $v(u') \leftarrow v^*(u' \mid \overline{\mathcal{B}}), \overline{v}(u') \leftarrow \overline{v}(u' \mid \overline{\mathcal{B}}), p(u') \leftarrow p^*(u' \mid \overline{\mathcal{B}})$
16:            if $v(u') + \overline{v}(u') > \underline{v}$ then
17:                add $u'$ to $Fringe$
18: return $(\underline{x}, \underline{v})$
```

Algorithm 6 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:     $pruned \leftarrow \emptyset$
5:     perform dominance pruning using Algorithm 7
6:     $L'_j \leftarrow L_j \setminus pruned$
7:     if $|L'_j| > W$ then
8:         restrict or relax the layer to get $W$ nodes with Algorithm 2
9:     $L_{j+1} \leftarrow \emptyset$
10:    for all $u \in L'_j$ do
11:        if $v^*(u \mid \mathcal{B}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}$ then        // RUB pruning
12:            continue
13:        for all $d \in \mathcal{D}_j$ do
14:            create node $u'$ with state $\sigma(u') = t_j(\sigma(u), d)$
               or retrieve it from $L_{j+1}$
15:            create arc $a = (u \xrightarrow{d} u')$ with $v(a) = h_j(\sigma(u), d)$ and $l(a) = d$
16:            add $u'$ to $L_{j+1}$ and add $a$ to $A$
17: merge nodes in $L_n$ into terminal node $t$
```

Algorithm 7 Dominance-based filtering of layer L_j of a DD $\mathcal{B}$.

```
1:  sort nodes $u$ in $L_j$ in *reverse* lexicographic order of $\Psi(u)$
2:  for all $u \in L_j$ do
3:      if $u$ is relaxed then
4:          continue
5:      if $Fronts_j.contains(\kappa(\sigma(u)))$ then
6:          $Front \leftarrow Fronts_j.get(\kappa(\sigma(u)))$
7:          $dominated \leftarrow False$
8:          for all $\Psi' \in Front$ do
9:              if $\Psi(u) \leq \Psi'$ then                // exit if $\Psi(u)$ is dominated
10:                 $dominated \leftarrow True$
11:                 break
12:             if $\Psi(u) \geq \Psi'$ then     // remove entries that $\Psi(u)$ dominates
13:                 $Front \leftarrow Front \setminus \{\Psi'\}$
14:         if $dominated$ then
15:             $pruned \leftarrow pruned \cup \{u\}$
16:         else                                  // add to front if non-dominated
17:             $Front \leftarrow Front \cup \{\Psi(u)\}$
18:     else                                     // initialize if first with given key
19:         $Front \leftarrow \{\Psi(u)\}$
20:         $Fronts_j.insert(\langle\kappa(\sigma(u)), Front\rangle)$
```

Algorithm 7 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 node dominates is removed from the *Front* with line 13, so that its size is kept as small as possible.

Note that this process is performed both for restricted and relaxed DDs. It means that both types of DD benefit from this filtering mechanism. In addition, the exploratory nature of restricted DDs can play an important role in quickly finding strong non-dominated nodes, as these will then impact the compilation of relaxed DDs and facilitate their divide-and-conquer role by pruning nodes that would otherwise have been explored.

Example 3.3.1. *Let us apply this procedure for layer L_3 of the exact DD given by Figure 3.1, assuming it has already been filled with the utilities of the nodes reached by the approximate DDs of Figures 2.2 and 2.3. Before starting the dominance check, we thus have that $Fronts_3 = \{\langle \mathbf{0}, \{(15,1),(14,3),(9,5),(3,9)\}\rangle\}$. We first compute the utility – given by $\Psi(u) = (v^*(u \mid \mathcal{B}), \sigma(u))$ – of each node in L_3: $\Psi(c_1) = (0,15), \Psi(c_2) = (6,11), \Psi(c_3) = (3,9), \Psi(c_4) = (12,7), \Psi(c_5) = (9,5)$ and $\Psi(c_6) = (15,1)$, and then order the nodes by reverse lexicographic order of those, which produces: $\{c_6, c_4, c_5, c_2, c_3, c_1\}$.*

- *The utility of node c_6 is already present in the front, so by definition c_6 cannot be dominated.*
- *Node c_4 is not dominated by any utility in the front and is thus added to the front. Moreover, it dominates the utility $(9,5)$ stored in the front, since $\Psi(c_4) = (12,7) \geq (9,5)$, so that previously non-dominated utility is removed from the front. After these two operations, we have: $Fronts_3 = \{\langle \mathbf{0}, \{(15,1),(14,3),(12,7),(3,9)\}\rangle\}$.*
- *The utility of node c_5 is equal to the one we just removed from the front, so it is dominated by node c_4 and added to the pruned set.*
- *Node c_2 is not dominated and dominates the utility $(3,9)$ since $\Psi(c_2) = (6,11) \geq (3,9)$. It thus replaces it in the front, which becomes: $Fronts_3 = \{\langle \mathbf{0}, \{(15,1),(14,3),(12,7),(6,11)\}\rangle\}$.*
- *Again, node c_3 has the same utility as the one we just removed from the front, so it is dominated by node c_2 and added to the pruned set.*
- *Finally, node c_1 is added to the front because it has the largest remaining capacity and is therefore non-dominated. The final front is given by $Fronts_3 = \{\langle \mathbf{0}, \{(15,1),(14,3),(12,7),(6,11),(0,15)\}\rangle\}$.*

3.4 Applications

This section details the DP models and dominance rules of the four optimization problems that will be used for our computational experiments, as well as the benchmark instances and the settings used in each case. Some of these problems are minimization problems, so a *rough lower bound* (RLB) will be described instead of an RUB. Moreover, whereas the given definition of the dominance utility assumes that *greater is better*, the opposite rule is applied when minimizing.

3.4.1 Traveling Salesman Problem with Time Windows

The *Traveling Salesman Problem with Time Windows* (TSPTW) is a variant of the well-known *Traveling Salesman Problem* (TSP) 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 $\mathcal{TW}_i = (e_i, l_i)$. The first customer is a dummy customer that represents the depot where the salesman must begin and end its tour. In this variant of the TSPTW, we are given an asymmetrical distance matrix D that contains in each entry (i, j) the distance D_{ij} separating customers i and j. In addition to the time windows controlling the earliest and latest time when the salesman can visit the customers, the horizon H limits the time at which the salesman must return to the depot. The objective is to find a tour that visits all customers during their time window and comes back to the depot at the earliest possible time. We thus consider the *makespan* variant of the problem, that looks to minimize the sum of the *travel time* and the *waiting time* accumulated when the salesman arrives early at some customers. However, only the transition value functions need to be adapted to solver the *travel time* objective.

3.4.1.1 Dynamic Programming Formulation

The DP model recalled here is the one presented in [Gil22], except for the transition functions for which additional checks are introduced. It is based on the DP model for the TSP from [HK62] that successively decides the next customer to visit, and defines states with the set of customers that still must be visited along with the current position of the salesman. For the TSPTW, this state representation is extended to a tuple $\langle L, t, M, P \rangle$, where L (locations) and t (time) respectively represent the set of locations where the salesman might be and the minimum time to reach one of these locations. The set M (must) contains the visits that must still be completed, while P (possible) are the visits that can possibly be done in addition to the ones in M. This set of possible visits P is useful for tightening the relaxation of the relaxed DD, as explained next. M and P are disjoint sets of customers that must or might be visited in order to close the tour even in case of relaxed states. We now describe the full DP model:

- Control variables: $x_j \in N$ with $j \in N$ decides which customer is visited in j-th position.
- State space: $\mathcal{S} = \{\langle L, t, M, P \rangle \mid L, M, P \subseteq N, M \cap P = \emptyset, 0 \leq t \leq H\}$. The root state is $\hat{r} = \langle \{0\}, 0, N, \emptyset \rangle$ and the terminal states are all states $\langle \{0\}, t, \emptyset, P \rangle$ with $0 \leq t \leq H$.

- Transition functions:

$$t_j(s^j, x_j) = \begin{cases} \left\langle t_j^L(s^j,x_j), t_j^t(s^j,x_j), t_j^M(s^j,x_j), t_j^P(s^j,x_j) \right\rangle, \\ \qquad\qquad \text{if } x_j \in s^j.M \text{ and } s^j.t + \min_{i \in s^j.L} D_{ix_j} \leq l_{x_j}, \\ \left\langle t_j^L(s^j,x_j), t_j^t(s^j,x_j), t_j^M(s^j,x_j), t_j^P(s^j,x_j) \right\rangle, \\ \text{if } x_j \in s^j.P \text{ and } s^j.t + \min_{i \in s^j.P} D_{ix_j} \leq l_{x_j} \text{ and } |M| < n - j, \\ \hat{0}, \qquad\qquad \text{otherwise.} \end{cases}$$

  where

$$\begin{aligned} t_j^L(s^j,x_j) &= \{x_j\} \\ t_j^t(s^j,x_j) &= \max\left\{e_{x_j}, s^j.t + \min_{i \in s^j.P} D_{ix_j}\right\} \\ t_j^M(s^j,x_j) &= s^j.M \setminus \{x_j\} \\ t_j^P(s^j,x_j) &= s^j.P \setminus \{x_j\} \end{aligned}$$

  Transitions are only allowed if they respect the time window constraint l_{x_j}. Since the salesman can be at multiple different positions in relaxed states, the minimum distance is used to compute the arrival time. The second condition in the transition functions ensures that no customers are selected from the possible set P when there remains enough stops only for the customers in the must-set M.

- Transition value functions: $h_j(s^j, x_j) = \min_{i \in s^j.L} D_{ix_j}$.

- Root value: $v_r = 0$.

3.4.1.2 Relaxation

The merging operator is defined as follows:

$$\oplus(\mathcal{M}) = \langle \oplus_L(\mathcal{M}), \oplus_t(\mathcal{M}), \oplus_M(\mathcal{M}), \oplus_P(\mathcal{M}) \rangle$$

where

$$\begin{aligned} \oplus_L(\mathcal{M}) &= \textstyle\bigcup_{s \in \mathcal{M}} s.L, \\ \oplus_t(\mathcal{M}) &= \min_{s \in \mathcal{M}} s.t, \\ \oplus_M(\mathcal{M}) &= \textstyle\bigcap_{s \in \mathcal{M}} s.M, \\ \oplus_P(\mathcal{M}) &= (\textstyle\bigcup_{s \in \mathcal{M}} s.M \cup s.P) \setminus (\textstyle\bigcap_{s \in \mathcal{M}} s.M). \end{aligned}$$

These operators ensure that all transitions are preserved and that the transition values are not increased. Indeed, the set of possible current location includes all possible locations that appear in the merged states. Similarly, the set of customers that must be visited is defined as the customers that must be visited in all states to merge, and the set of customers that can possibly be visited as those that can be visited in some states to merge. Note that the M and P sets are disjoint so that P contains customers that must or might be visited in some states, but no customers that must be visited in all states. Finally, the merging operator optimistically keeps the minimum time among the states to merge. The relaxed transition value operator is simply the identity function $\Gamma_{\mathcal{M}}(v, u) = v$.

3.4.1.3 Rough Lower Bound

The RLB shown in Equation (3.1) computes an estimate of the remaining distance to cover in order for the salesman to finish his tour and return to the depot. For all customers i in M, the distance of the cheapest edge incident to i is added. Then, $n - j - |M|$ customers from P must be visited in order to complete the tour. Since this is a lower bound computation, we create a vector C containing the cheapest edge incident to each customer in P and select the $n - j - |M|$ shortest ones. We denote by $X_{<,i}$ the i-th smallest element from a vector X.

$$\underline{v}_{rlb}(s^j) = \sum_{i \in s^j.M} cheapest_i + \sum_{i=1}^{n-j-|M|} C_{<,i} \tag{3.1}$$

A check is also added to detect states where it is impossible to visit independently all of the customers from M and at least $n - j - |M|$ customers from P, given the current time and the time windows of the remaining customers. Plus, if the estimate provided by Equation (3.1) prevents the salesman from returning to the depot within the time constraint, a RLB of ∞ is returned, effectively discarding the corresponding states.

3.4.1.4 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: $\kappa(s) = (s.L, s.M, s.P)$. Note that as the dominance rule is only applied to exact nodes, we will always have that $s.M$ is a singleton and $s.P$ is empty. 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 first operator is also valid for the *travel time* version of the TSPTW.

3.4.1.5 Experimental Setting

All configurations of the DD-based solver were tested on a classical set of benchmark instances introduced in the following papers [Asc96; Dum+95; Lan+93; Pes+98; PB96]. 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.

3.4.2 Aircraft Landing Problem

The *Aircraft Landing Problem* (ALP) requires to schedule the landing of a set of aircraft $N = \{0, \dots, n-1\}$ on a set of runways $R = \{0, \dots, r-1\}$. The aircraft have a target time T_i that gives the earliest landing time, and latest landing time L_i. Moreover, the set of aircraft 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 a feasible schedule that contains all the aircraft and minimizes the total waiting time of the aircraft – the delay between their target time and scheduled landing time – while respecting their latest landing time.

3.4.2.1 Dynamic Programming Formulation

We reproduce here the DP model presented in [LBS15] – except for the RLB introduced this section and a slightly different dominance rule – where 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. To initialize the root state and to formulate the relaxation, 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: valid states are contained in the following state space:

$$\begin{aligned} \mathcal{S} = \{(Q, ROP) \mid & \forall i \in C : 0 \leq Q_i \leq |A_i|, \\ & \forall k \in R : ROP_k.l \geq 0, ROP_k.c \in C \cup \{\perp\}\}. \end{aligned}$$

  The root state is $\hat{r} = (\langle |A_0|, \dots, |A_{c-1}| \rangle, \langle (0, \perp), \dots, (0, \perp) \rangle)$ and the terminal states are of the form $(\langle 0, \dots, 0 \rangle, ROP)$.

- Transition functions: if A_i^k gives the aircraft from class i that must be scheduled when there are k aircraft 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:

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

The first case concerns a runway where no aircraft has been scheduled yet, and therefore uses the aircraft target time as the earliest landing time. The second case deals with runways having an aircraft previously scheduled and that must respect the separation time between the two consecutive aircraft. Lastly, if the landing of an aircraft was scheduled on the runway but its class is unknown, the minimum separation time before making an aircraft of class x land is used in the formula. This allows us to define the transition functions as:

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

where

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

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

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 aircraft when there are no aircraft left to schedule. The state is updated by decrementing the remaining number of aircraft to schedule for the selected class, and by storing the time and aircraft class of the scheduled landing.

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

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

- 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.

3.4.2.2 Relaxation

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 aircraft for each class is stored in the merged state:

$$\oplus_Q(\mathcal{M}) = \left\langle \min_{s \in \mathcal{M}} s.Q_0, \dots, \min_{s \in \mathcal{M}} s.Q_{c-1} \right\rangle .$$

For the ROP, the minimum latest landing time on each runway is kept and the last aircraft classes scheduled are reset to $\perp$:

$$\oplus_{ROP}(\mathcal{M}) = \left\langle (\min_{s \in \mathcal{M}} s.ROP_0.l, \perp), \dots, (\min_{s \in \mathcal{M}} s.ROP_{r-1}.l, \perp) \right\rangle .$$

The relaxed transition value operator is the identity function $\Gamma_{\mathcal{M}}(v, u) = v$.

3.4.2.3 Rough Lower Bound

Assuming the separation matrix respects the triangle inequality, if for a given state, the next aircraft of any class cannot be scheduled on time on any of the runway, then this state cannot lead to feasible solutions. Indeed, if the triangle inequality is verified, the earliest landing time of the aircraft can only be delayed by other transitions. If such scenario occurs, an RLB of ∞ can be returned to prune the state.

3.4.2.4 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: $\kappa(s) = (s.Q, \langle s.ROP_0.c, \dots, s.ROP_{r-1}.c \rangle)$ and $\psi(s) = \langle s.ROP_0.l, \dots, s.ROP_{r-1}.l \rangle$.

3.4.2.5 Experimental Setting

A set of 720 random instances was generated, with the number of aircraft given by $n \in \{25, 50, 75, 100\}$, the number of runways by $r \in \{1, 2, 3, 4\}$, and a fixed number of aircraft classes $c = 4$. The target landing times were generated according to a Poisson arrival process with a mean inter-arrival time of $40/r$, instances with more runways thus require producing denser schedules. The separation matrices were created by sampling $K \in \{2, 3, 4\}$ reference points uniformly in a 2-dimensional spaces with bounds $[0, 100]^2$. Each aircraft class $i \in C$ is then associated with one of these reference points (x, y) – the aircraft classes are distributed equally among the reference points – and assigned a neighboring point $p_i = (x + \Delta_x, y + \Delta_y)$ with Δ_x and Δ_y sampled according to a Gaussian distribution of mean $\mu = 0$ and standard deviation $\sigma \in \{10, 20, 30\}$. The separation matrix is then obtained by computing the Manhattan distance $S_{i,j} = |p_i.x - p_j.x| + |p_i.y - p_j.y|$ between each pair of points p_i, p_j with $i, j \in C$ and $i \neq j$. Finally, since realistically the separation time between two aircraft of the same class is non-zero, it is arbitrarily set to $S_{i,i} = \frac{3}{4} \min_{j \in C \setminus \{i\}} S_{i,j}$. A fixed width of $W = 100$ is used for all experiments concerning the ALP.

3.4.3 Longest Common Subsequence Problem

In the *Longest Common Subsequence Problem* (LCS), we are given a set of m input strings $S = \{S_0, \ldots, S_{m-1}\}$ composed of characters from a given alphabet Σ. The goal is to find the longest *subsequence* appearing in all strings, that can be obtained by removing some characters from each input string. With n is defined as $n = \max\{|S_i| \mid S_i \in S\}$, there exists a well-known DP model for solving this problem with complexity $O(n^m)$ [Gus97]. It was already employed in a DD-based formulation in [HR21] for generating tight upper bounds with relaxed DDs. This section reproduces this formulation almost identically, but using notations consistent with the other DP models covered.

3.4.3.1 Dynamic Programming Formulation

Before specifying the DP model, we define additional matrices that simplify the formulas, and also reduce the complexity of the operators if they are precomputed. The $(|S_i| + 1) \times |\Sigma|$ matrices N_i for each string S_i are defined such that $N_i^{c,j}$ gives the position of the next character c occurring in string i after position j. They can be computed by applying the following backward recurrence for each $i = 0, \dots, m-1$, $c \in \Sigma$ and $j = |S_i|, \dots, 0$:

$$N_i^{c,j} = \begin{cases} \bot, & \text{if } j = |S_i|, \\ j, & \text{if } S_i[j] = c, \\ N_i^{c,j+1}, & \text{otherwise.} \end{cases}$$

Similarly, the $(|S_i| + 1) \times |\Sigma|$ matrices $R_i^{c,j}$ gives the remaining number of occurrences of character c in string i after position j, and can be computed by applying the following backward recurrence for each $i = 0, \dots, m-1$, $c \in \Sigma$ and $j = |S_i|, \dots, 0$:

$$R_i^{c,j} = \begin{cases} 0, & \text{if } j = |S_i|, \\ R_i^{c,j+1} + 1, & \text{if } S_i[j] = c, \\ R_i^{c,j+1}, & \text{otherwise.} \end{cases}$$

Since the length of the LCS is bounded by the length of the shortest string in S, we assume that the strings are ordered by increasing length $|S_0| \leq |S_1| \leq \dots \leq |S_{m-1}|$ and therefore reason about the characters contained in S_0. Indeed, when constructing a subsequence from left to right while keeping track of the current position in each string with a vector $\langle p_0, \dots, p_{m-1} \rangle$, if a character c of string S_0 is appended to the subsequence, it suffices to select the leftmost occurrence of c after position p_i in each string S_i. Therefore, we define $\underline{n} = |S_0| = \min \{|S_i| \mid S_i \in S\}$. Given those additional definitions, the DP model for the LCS can be formulated as follows.

- Control variables: a first possibility is to use variables $x_j \in \{0, 1\}$ with $j = 0, \dots, n' - 1$ that decide whether the character at position j of string S_0 is added to the subsequence. However, a more compact model can be obtained by defining alternative variables x'_j that decide which character is appended to the subsequence, or whether the subsequence is ended. Formally, this is modeled by using long arcs, meaning that with variable $x'_j \in \Sigma \cup \{\text{END}\}$, the assignment $x'_j = c$ corresponds to $(x_j, \dots, x_{p'}) = (0, \dots, 0, 1)$ with $p' = N_0^{c,j}$, i.e. all characters between positions j and $p' - 1$ of string S_0 are ignored and the first occurrence of character c after position j is selected.

- State space: the states contain the current position in each string $\mathcal{S} = \{\langle p_0, \dots, p_{m-1} \rangle \mid 0 \leq p_i \leq |S_i|\}$. The root state is $\hat{r} = \langle 0, \dots, 0 \rangle$ and the terminal stated are of the form $\langle |S_0|, p_1, \dots, p_{m-1} \rangle$.

- Transition functions:

$$t_j(s^j, x_j) = \begin{cases} \left\langle N_0^{c,s_0^j} + 1, \dots, N_{m-1}^{c,s_{m-1}^j} + 1 \right\rangle, & \text{if } x_j \in \Sigma \text{ and} \\ \qquad R_i^{c,s_i^j} > 0, \forall i = 0, \dots, m-1, \\ \langle |S_0|, \dots, |S_{m-1}| \rangle, & \text{if } x_j = \text{END}, \\ \hat{0}, & \text{otherwise.} \end{cases}$$

  In the above equation, a transition with character c is allowed if each string S_i still contains one occurrence of it after position s_i^j. Alternatively, any state can transition to the terminal state by deciding to end the subsequence.

- Transition value functions: the objective value is incremented each time a character is appended to the subsequence, i.e.:

$$h_j(s^j, x_j) = \begin{cases} 1, & \text{if } x_j \in \Sigma, \\ 0, & \text{otherwise.} \end{cases}$$

- Root value: $v_r = 0$.

3.4.3.2 Relaxation

A valid relaxation can be defined by computing the minimum current position in each string among the states to merge. The merging operator is thus: $\oplus(\mathcal{M}) = \langle \min_{s \in \mathcal{M}} s_0, \dots, \min_{s \in \mathcal{M}} s_{m-1} \rangle$. The relaxed transition value operator is the identity function $\Gamma_{\mathcal{M}}(v, u) = v$.

3.4.3.3 Rough Upper Bound

The RUB used in [HR21] combines two optimistic estimates of the number of characters that can still be added to the subsequence for a given state. The first one simply sums the minimum remaining number of occurrences of each character among all strings: $UB_1(s^j) = \sum_{c \in \Sigma} \min_{i=0,\dots,m-1} R_i^{c,j}$. The second one is based on the fact that the LCS can be solved very efficiently with DP for $m = 2$. Therefore, the LCS of each consecutive pair of strings can be precomputed and the corresponding DP table stored. An estimate can then be obtained by finding the length of the shortest LCS among the consecutive pairs of strings, given the current position in each string: $UB_2(s^j) = \min_{i=0,\dots,m-2} LCS(S_i[j:], S_{i+1}[j:])$. The final RUB is given by the tighter estimate: $\bar{v}_{rub}(s^j) = \min\left\{UB_1(s^j), UB_2(s^j)\right\}$.

3.4.3.4 Dominance Rule

Given states with the same current position in string S_0 – i.e. at the same DD layer. It is always better to have both lower positions and a greater objective value. This is expressed by the following dominance key and dominance utility vector: $\kappa(s) = \mathbf{0}$ and $\psi(s) = s$.

3.4.3.5 Experimental Setting

We used the following classical benchmark instances for the LCS: BB [BB07], BL [BF16], Rat, Virus and Random [ST09], Poly and Abstract [Nik+21], but limited to instances with $m < 10$. A fixed width of 100 is used for all experiments concerning the LCS. Moreover, the *Pooled* DD implementation described in [Gil22] is used to compile DDs with long arcs.

3.4.4 0–1 Knapsack Problem

The KP is equivalent to the BKP when all items have unit quantities. We thus solve the KP with all the ingredients presented in Chapters 2 and 3. In addition, variables are ordered so that the items are considered in decreasing *profit-to-weight* ratios and the LP bound of [Dan57] is used as the RUB. A set of benchmark instances constituting of a random selection of 2% of the instances from [Pis05] (636 instances) and 10% of the instances from [SCM21] (530 instances). Again, a fixed width of 100 is used for all experiments concerning the KP.

3.5 Computational Experiments

In this section, we wish to experimentally evaluate the impact of the integration of dominance rules within the DD-based optimization approach. To this end, the four DP formulations described in Section 3.4 were all implemented through the modeling interfaces of DDO [GSC21] and were applied to solve the associated benchmark instances. We also performed experiments for the TSPTW with the MIP model referred to as Formulation (1) in [HT18]. It is solved with Gurobi 9.5.2 [Gur22] using a single thread and otherwise default settings. DDO is also executed on a single thread and uses the *MinLP* node selection heuristic. For all problems, the solvers were given 600 seconds to solve each instance to optimality. These configurations will remain fixed for all computational experiments presented in this thesis. In the experiments of this particular chapter, DDO always employs LEL cutsets. The server used to run all the experiments of this thesis is equipped with Intel Xeon Platinum 8160 processors with base frequency 2.10GHz. Moreover, the code is available on GitHub at `https://github.com/vcoppe/ddo/tree/thesis_xp`.

3.5.1 Number of Benchmark Instances Solved

In the following, we will refer to the classical B&B algorithm as DDO, and when exploiting dominance rules as DDO+D. Figure 3.2 shows the number of instances solved by each solver and configuration with respect to the solving time. For all four problems considered, DDO+D solves more instances than DDO, and by a large margin except for the KP. 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. For the TSPTW, DDO and DDO+D perform respectively slightly and far better than Gurobi, which underlines the strength of a recursive formulation for such a constrained problem.

3.5.2 Number of Node Expansions

The impact of the dominance rules can also be measured in terms of the total number of nodes expanded during the successive DD compilations. This measure accounts for all nodes expanded during top-down compilation of both restricted and relaxed DDs. This is represented for each problem by Figure 3.3, which exhibits very similar trends to Figure 3.2. Indeed, it appears that the increase in the number of instances solved within the time limit when comparing DDO and DDO+D is caused by a decrease in the number of node expansions needed to solve each given instance. 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, and there can be many of them for instances with many different unique item weights.

3.5.3 Quality of the First Solution

Another important dimension for a solver is the quality of the first solution found, which captures its *anytime* behavior. We expect it to be improved when exploiting dominance rules as they are able to filter many less promising solutions during the top-down compilation of restricted DDs. Figure 3.4 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

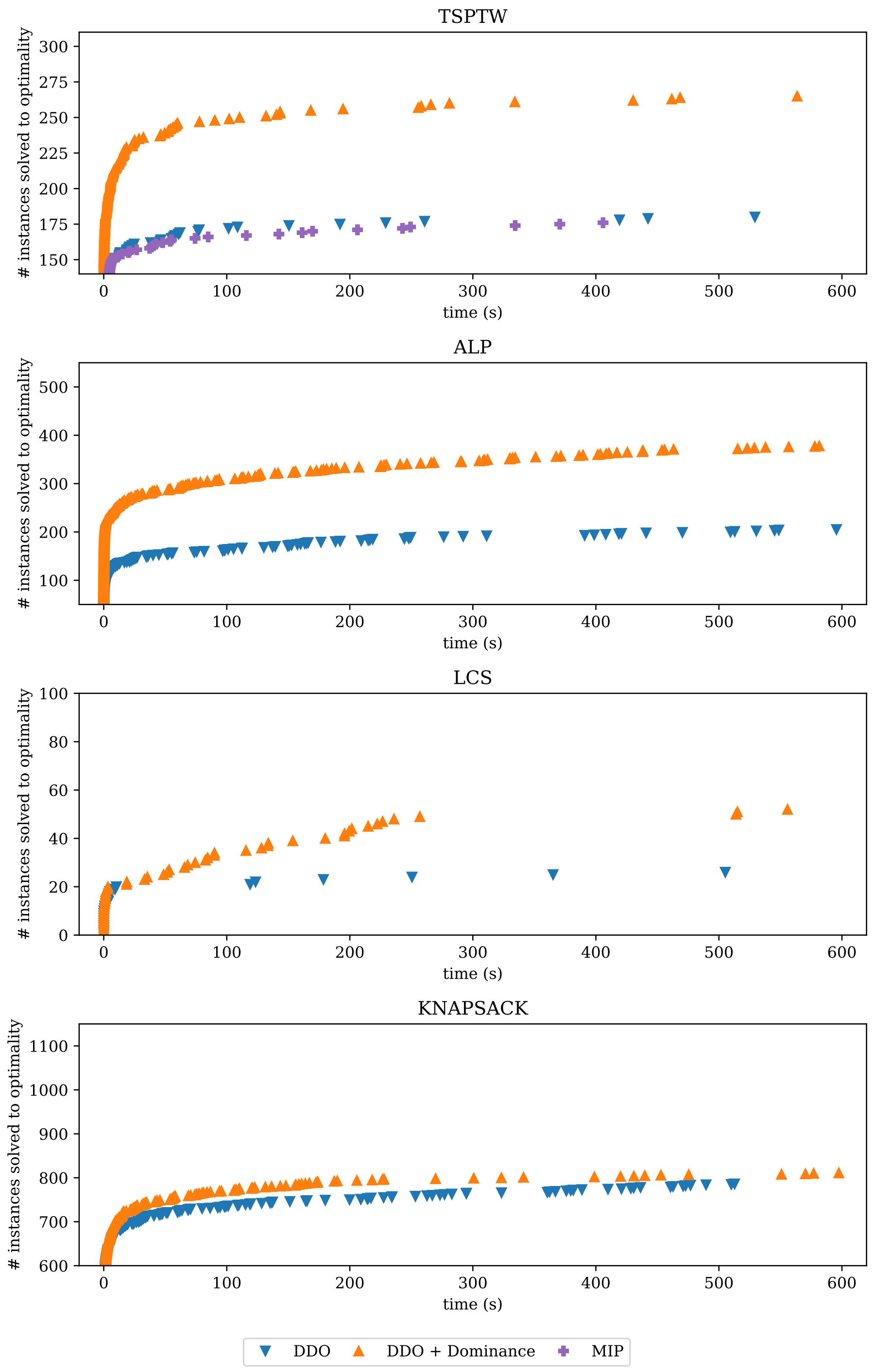


Figure 3.2: Number of instances solved over time by DDO, DDO+D and Gurobi for four different problems.

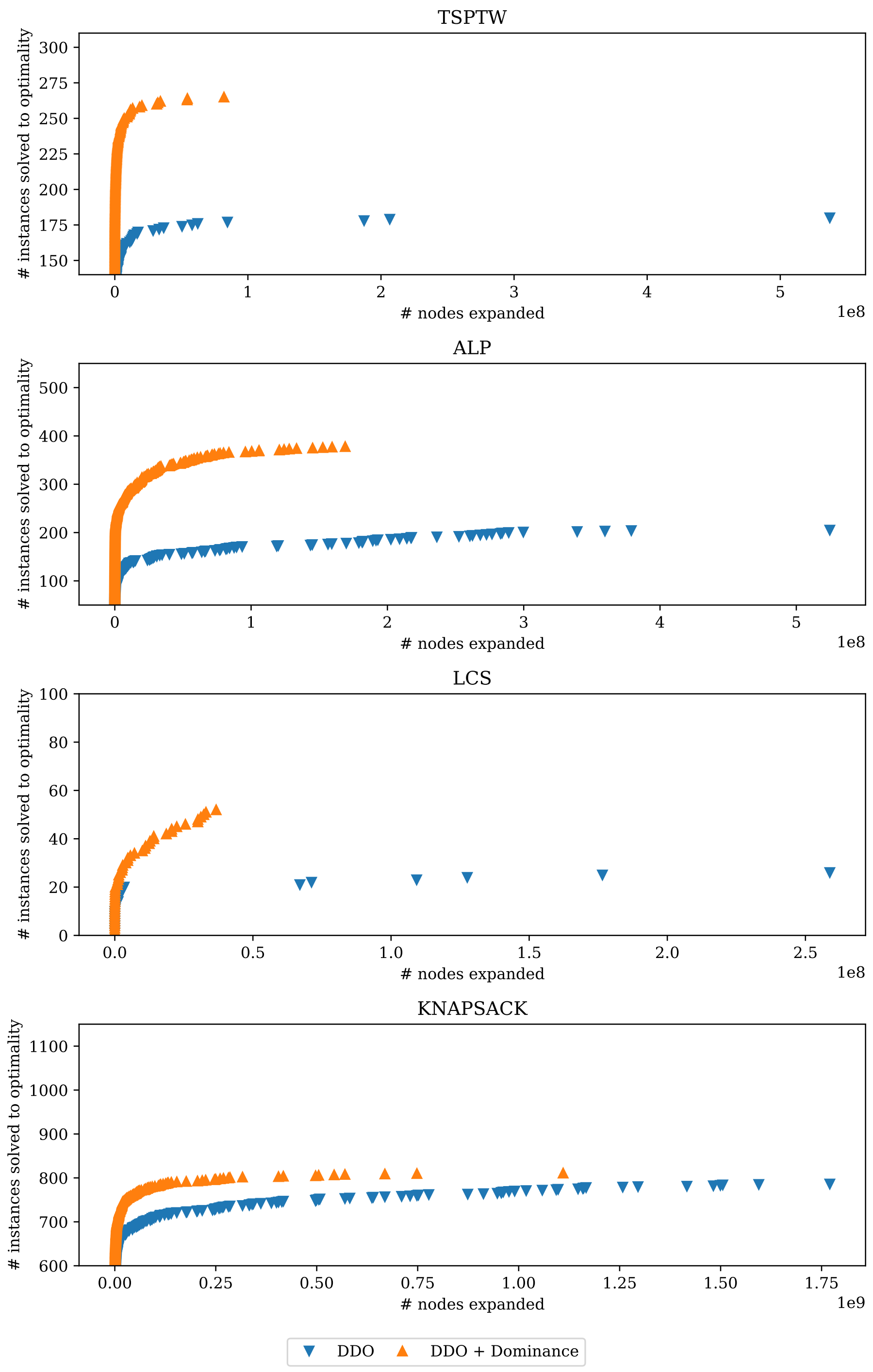


Figure 3.3: Number of instances solved by DDO and DDO+D with respect to the number of DD nodes expanded for four different problems.

solution is found for the TSPTW and the ALP. Indeed, those problems both have time window constraints and therefore restricted DDs might not produce feasible solutions at each compilation. This comparison allows us to make several observations for each problem:

- TSPTW: the quality of the first solution found is only moderately impacted by the addition of dominance rules. DDO+D finds a better first solution than DDO in 26 cases, against 7 cases in the other direction. However, 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 many instances, and much earlier for a few of them. DDO+D obtains a solution earlier than DDO for 42 instances, whereas the opposite is true for only 3 instances.

- ALP: we can observe that the quality of the first solution found by DDO+D is in general slightly better than the one obtained by DDO when both configurations find a feasible solution. Furthermore, for 197 instances, DDO does not manage to find a single feasible solution to the problem whereas DDO+D finds one for 69 of those – this does not include some instances that are actually infeasible, which is proven by either configuration for 127 instances. In addition, when comparing the iteration at which the first solution is found, it appears that DDO+D finds many of them at the first iteration and most of them within 20000 iterations. On the other hand, DDO needs up to 60000 iterations to find a first solution for instances where DDO+D found a solution after just a few iterations, or simply fails to do so in many cases.

- LCS and KP: they are both maximization problems, so this time DDO+D compares better for data points located above the diagonal line. This occurs for 201 LCS instances, whereas DDO finds a better first solution in 93 cases. Although it is difficult to distinguish the solution values for the KP on Figure 3.4, DDO+D actually finds a slightly better solution than DDO in 230 cases, compared to only 20 cases in the opposite direction.

Integrating dominance rules therefore not only accelerates the B&B algorithm, but 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. This is a very important improvement for applications that use restricted DDs only as a primal heuristic, e.g. [OH19].

3.5.4 Memory Consumption

To conclude this experimental analysis, we discuss the memory footprint of maintaining the *Fronts* used to continually derive dominance relations with

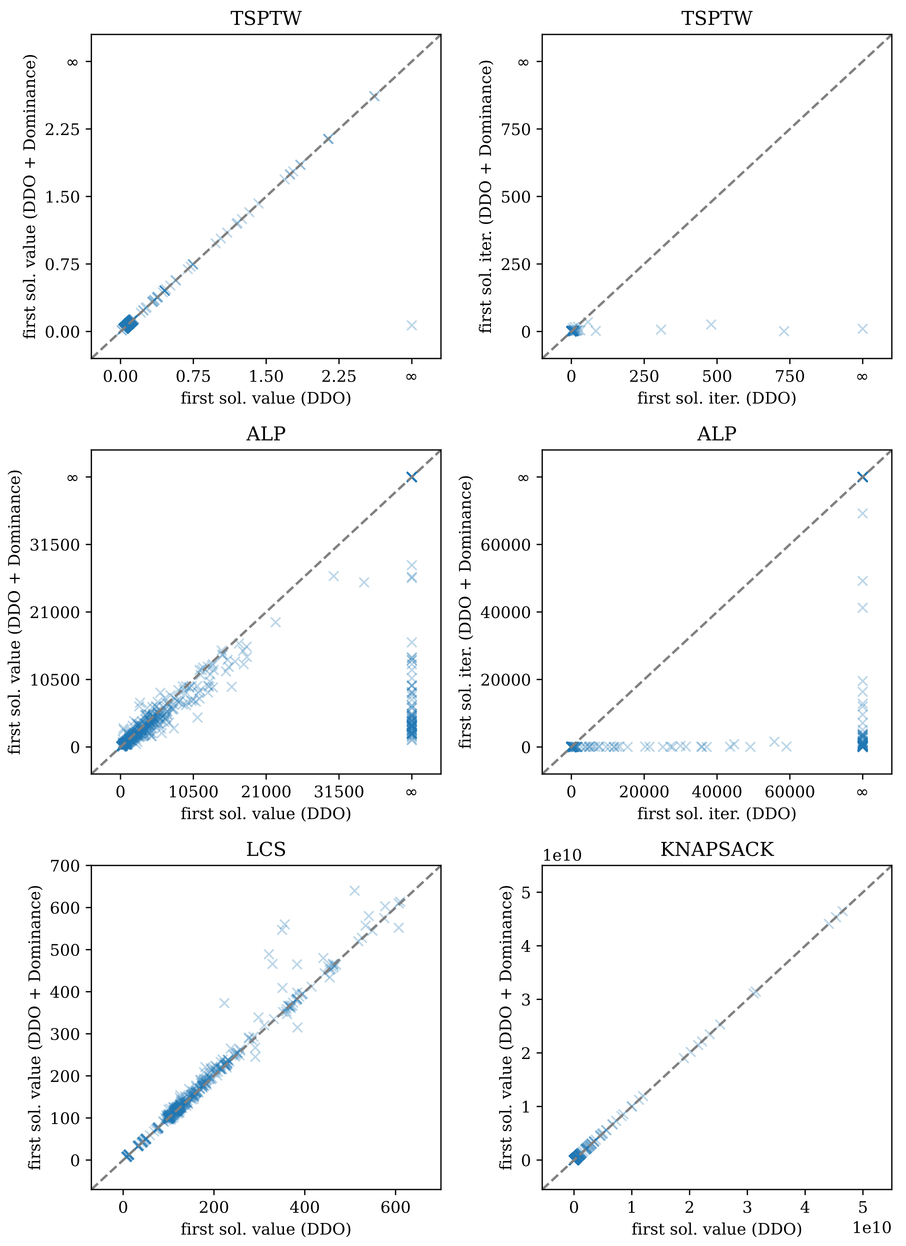


Figure 3.4: Comparison of the value of the first solution obtained for each instance by DDO and DDO+D, and of the iteration at which this solution is found for TSPTW and ALP.

respect to non-dominated exact nodes previously found. Figure 3.5 represents the number of instances solved by each configuration for each problem, with respect to the peak amount of memory used during the search. For all of them, we observe that DDO solves more instances than DDO+D for very low amount of memory. When both configurations exceed respectively around 3MB, 7MB and 50MB for the TSPTW, ALP and LCS, DDO+D starts solving more instances with the same amount of memory – under the given time limit. Therefore, storing non-dominated nodes ends up reducing the size of DDs, of their ECs and of the *Fringe* in such a way that DDO+D uses less memory to solve some instances. For the KP, both configurations solve approximately the same number of instances when the peak memory is above 100MB. Even if the dominance rules are less impactful for this problem, they do not penalize the algorithm in terms of memory consumption.

3.6 Conclusion

In this chapter, we provided a formalism for specifying dominance rules of DP models. We also explained how they can be exploited within the DD compilation algorithm, as well as for the B&B algorithm as whole by introducing a persistent data structure used to detect dominance relations across DD compilations. The modeling of the dominance rules was illustrated on four optimization problems and its impact was evaluated through extensive computational experiments. The results clearly highlighted the interest of this additional ingredient, which significantly reduced 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 a large number of instances previously unsolved by DDO. Moreover, the experiments demonstrated 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, we discussed the memory footprint of implementing such an exhaustive dominance checking strategy. It appears that the extra memory required by the *Fronts* largely compensated by its filtering capacity that eliminates redundant parts of the search space.

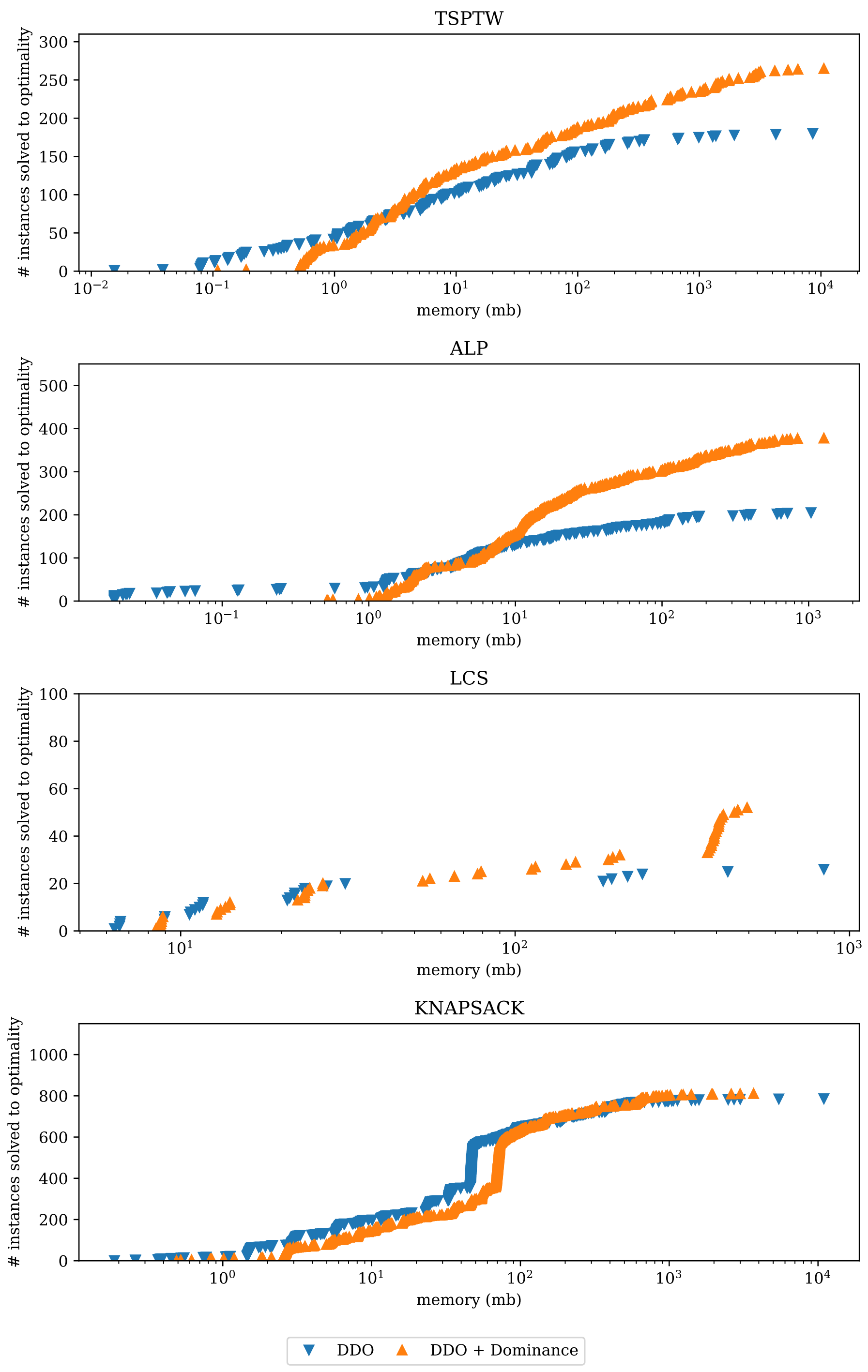


Figure 3.5: Number of instances solved by DDO and DDO+D with respect to the peak amount of memory used for four different problems.

Caching | 4

This chapter is largely based on the following paper currently under review: V. Coppé, X. Gillard, and P. Schaus. "Decision Diagram-Based Branch-and-Bound with Caching for Dominance and Suboptimality Detection". In: (2023). arXiv: 2211.13118. It presents a caching mechanism that collects expansion thresholds throughout the B&B algorithm, which are used to avoid expanding multiple nodes with the same DP state when proved unnecessary. This chapter presents an extended version of the approach that allows it to coexist with the dominance rules of Chapter 3.

4.1 Introduction

The contributions of this chapter stem from a simple observation: as opposed to classical B&B for *mixed-integer programming* (MIP), the DD-based B&B does not split the search space into disjoint parts. The reason is that in this framework, DDs are based on a DP formulation of the discrete optimization problem at hand, and such kind of model typically contains many overlapping subproblems. As a result, the B&B may explore some subproblems multiple times. In addition, the approximate DDs compiled during the algorithm can repeat a lot of the work done previously because they cover overlapping parts of the search space. This inability to build on previous computational efforts is unfortunate, for that is the very strength of the DP paradigm. A first attempt to address this problem was made in [CGS22] by changing the ordering of the nodes in the B&B. By employing a breadth-first ordering, it is guaranteed that at most one B&B node corresponding to each given subproblem will be explored. However, this approach sacrifices the benefits of best-first search – always exploring the most promising node to try to improve the incumbent solution and tighten the bounds at the same time – and leaves the issue of overlapping approximate DDs unsolved.

In this chapter, the same problem is tackled while allowing a best-first ordering of the B&B nodes. In the same fashion that a closed list prevents the re-expansion of nodes in shortest-path algorithms, we propose to maintain a

Cache that stores an *expansion threshold* for each DP state reached by an exact node of any relaxed DD compiled during the B&B. These expansion thresholds combine *dominance* and *pruning thresholds* that exploit the information contained in relaxed DDs to the fullest extent. The former detect *path dominance* relations between partial solutions – a subset of the dominance relations covered in Chapter 3 – while the latter extrapolate the pruning decisions related to nodes filtered by the RUBs and LocBs introduced by [Gil+21], and by the dominance checks described in Chapter 3. By consulting the expansion thresholds stored in the *Cache*, many partial solutions are guaranteed to be either dominated by other partial solutions previously found, or suboptimal with respect to bounds already obtained, and can therefore be pruned during the compilation of subsequent approximate DDs.

In the same line of research, [RCR22; RCR23] suggested inserting open relaxed DDs directly in the B&B queue instead of open nodes, and described a *peeling* operation that splits a relaxed DD into two parts: the first containing all the paths traversing a chosen node and the second containing the rest. The peeling operation permits two things when used as a replacement for classical branching procedures: it allows warm-starting the compilation of subsequent relaxed DDs, and it strengthens the bounds of the relaxed DD on which the peeling was conducted. This mechanism helps in reducing the overlap between the approximate DDs compiled during the search but does not fully address the issue. These improvements could thus be combined with the techniques hereby introduced.

This chapter begins with a discussion of the caveats that we try to address in Section 4.2. Next, Section 4.3 presents the expansion thresholds through its two components – the dominance and pruning thresholds – as well as their computation and integration into both the B&B algorithm and the approximate DD compilation procedure. Section 4.4 then discusses the limitations of the techniques introduced in the chapter. Finally, experimental results on six different discrete optimization problems are reported and discussed in Section 4.6 before concluding.

4.2 Caveats of DD-based Branch-and-Bound

As explained in Chapter 2, DD-based B&B enables solving discrete optimization problems by taking advantage of the compactness of DP models. Nevertheless, a few observations suggest that all the properties of this type of model have not yet been exploited. First, we point out that branching does not split the search space into disjoint parts. Indeed, the very nature of DP models is the ability to solve a large problem by recursively dividing it into smaller *overlapping* subproblems. Yet, vanilla DD-based B&B processes subproblems as if they were completely independent. To give a better intuition of why this might cause the algorithm to waste computational effort, we propose to look at DD-based B&B as a classical shortest-path algorithm performed on the graph induced by a DP model – in case of directed acyclic graphs, the shortest-path and longest-path problems are equivalent. DD-based B&B can be seen as a combination of *best-first* search at the B&B level with a sort of *breadth-first* search up to the EC of relaxed DDs, coupled with dedicated bounding and pruning procedures. However, while it shares many similarities with shortest-path algorithms like A* [HNR68], it lacks one of their important ingredients: the *closed list.* This data structure collects nodes expanded throughout the search and is used to check whether a given node was already expanded, to avoid expanding it again or adding it to the set of open nodes.

Due to the absence of such data structure, both levels of the B&B algorithm end up performing some computations multiple times. At the B&B level, nothing prevents from adding multiple nodes with the same DP state to the *Fringe* of Algorithm 4 nor triggering the compilation of approximate DDs for several of them at different stages of the search. Moreover, at the lower level, the approximate DDs can significantly overlap, even when rooted at nodes associated with different DP states. For instance, suppose we would compute exact DDs for the two subproblems contained in the FC of Figure 2.3, respectively rooted at a_1 and a_2. As one can observe on Figure 2.1, they respectively contain 18 and 15 nodes, among which 11 nodes appear in both DDs. Therefore, if these two DDs are compiled successively within the B&B, most of the work done for the first DD is repeated to compile the second one while many transitions could actually yield a value worse or equal than the one obtained before. In general, the structure of DP state transition systems causes such scenarios to occur very frequently.

The aim of this chapter is thus to integrate some form of closed list into DD-based B&B in order to mitigate the amount of duplicate computations at both levels of the algorithm. Unfortunately, a simple closed list cannot be used since DP states are not expanded purely in best-first order. In other words, given a relaxed DD $\overline{\mathcal{B}}$ and an exact node $u \in \overline{\mathcal{B}}$, nothing guarantees that $p^*(u \mid \overline{\mathcal{B}})$ – the longest path to reach u within $\overline{\mathcal{B}}$ – is the longest path to reach a node with state $\sigma(u)$ within the complete exact DD.

4.3 Branch-and-Bound with Caching

In this section, we explain how the caveats mentioned in Section 4.2 can be addressed. Our idea is to augment DD-based B&B with a caching mechanism that associates each DP state to an *expansion threshold* that conditions the need for future expansion of nodes with the same DP state. Formally, after compiling a relaxed DD $\overline{\mathcal{B}}$, an expansion threshold $\theta(u \mid \overline{\mathcal{B}})$ can be derived for each exact node $u \in \overline{\mathcal{B}}$ by exploiting the information contained in $\overline{\mathcal{B}}$. It is then stored as a *key-value* pair $\langle \sigma(u), \theta(u \mid \overline{\mathcal{B}}) \rangle$ in an associative array referred to as the *Cache*. When compiling a subsequent approximate DD $\mathcal{B}$, if a node u' happens to be generated such that $\sigma(u')$ corresponds to an existing key in the *Cache*, the expansion of node u' is skipped if its longest path value $v^*(u' \mid \mathcal{B})$ is not strictly greater than the expansion threshold stored. This filtering is carried out when calling Algorithm 10 within the DD compilation procedure of Algorithm 9.

Concerning the value of these expansion thresholds, a classical DP caching strategy would keep track of the longest path value obtained for each DP state, i.e. by having $\theta(u \mid \overline{\mathcal{B}}) = v^*(u \mid \overline{\mathcal{B}})$ for each exact node $u \in \overline{\mathcal{B}}$. In this section, we show how stronger expansion thresholds can be obtained by exploiting all the information contained in relaxed DDs. To give a better intuition on the origin of the expansion thresholds, we split their computation into two distinct thresholds that each cover a specific scenario requiring the expansion node associated with an already visited DP state. The *dominance* thresholds detect path dominance relations between partial solutions found throughout the search, while the *pruning* thresholds extrapolate successful past pruning decisions. Dominance thresholds can work alone if the RUB, LocB and dominance pruning rules are not used, but must be complemented by pruning thresholds whenever these extensions are involved. In the general case, they are thus combined into a single expansion threshold that covers both scenarios concomitantly.

After giving a formal definition of the aforementioned thresholds, we explain how they can be computed efficiently for all exact nodes visited during the compilation of relaxed DDs before being stored in the *Cache*. Finally, we describe how the B&B and the compilation of approximate DDs can be modified to benefit from the expansion thresholds.

4.3.1 Dominance Thresholds

As explained in Section 2.4, the compilation of a relaxed DD $\overline{\mathcal{B}}$ rooted at node u_r allows decomposing the corresponding subproblem into a set of subproblems given by nodes in $EC(\overline{\mathcal{B}})$. By extension, any exact node $u_1 \in \overline{\mathcal{B}}$ is also decomposed into a set of subproblems, given by nodes in $EC(\overline{\mathcal{B}})$ that belong to its successors $Succ(u_1 \mid \overline{\mathcal{B}})$. For example, an exact decomposition of node c_2 of Figure 4.1(a) is given by $Succ(c_2 \mid \overline{\mathcal{B}}) \cap EC(\overline{\mathcal{B}}) = \{d_1\}$. Among these nodes, only those that pass the pruning test at line 15 of Algorithm 4 are then added to the *Fringe* of the B&B. For simplicity, let us denote this set of nodes by

$$Leaves_d(u_1 \mid \overline{\mathcal{B}}) = \left\{ u \in Succ(u_1 \mid \overline{\mathcal{B}}) \cap EC(\overline{\mathcal{B}}) \mid v^*(u \mid \overline{\mathcal{B}}) + \overline{v}(u \mid \overline{\mathcal{B}}) > \underline{v} \right\}$$

where $\underline{v}$ is the value of the incumbent solution. Because we are solving optimization problems, we are only interested in one of the longest $r \rightsquigarrow u_2$ paths for each node $u_2 \in Leaves_d(u_r \mid \overline{\mathcal{B}})$, and other paths with lower or equal value can safely be ignored. As a result, we can infer *path dominance* relations between the paths obtained within relaxed DDs, and use them later to detect non-dominated paths that are thus still relevant to find the optimal solution of the problem.

Definition 4.3.1 (Path dominance). *Given a $r \rightsquigarrow u_1$ path p_1 and a $r \rightsquigarrow u_2$ path p_2 such that $\sigma(u_1) = \sigma(u_2)$, p_1 is said to dominate p_2 – formally denoted $p_1 \succ p_2$ – if and only if $v(p_1) > v(p_2)$. If $v(p_1) \geq v(p_2)$, then p_1 weakly dominates p_2, expressed as $p_1 \succeq p_2$.*

Path dominance thus corresponds to node dominance in the case where the two nodes considered have the same DP state. Given a relaxed DD $\overline{\mathcal{B}}$ and three exact nodes $u_1 \in \overline{\mathcal{B}}$, $u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}})$ and $u'_1 \notin \overline{\mathcal{B}}$ such that $\sigma(u'_1) = \sigma(u_1)$. The first scenario that requires the expansion of node u'_1 happens when it is possible that the concatenation $p_1 \cdot p_2$ of a prefix path $p_1 : r \rightsquigarrow u'_1$ with a suffix path $p_2 : u_1 \rightsquigarrow u_2$ dominates the longest path $p^*(u_2 \mid \overline{\mathcal{B}})$ found for the cutset node u_2 during the compilation of $\overline{\mathcal{B}}$. Formally, the dominance relation imposes that u'_1 be expanded if $p_1 \cdot p_2 \succ p^*(u_2 \mid \overline{\mathcal{B}})$. This property leads to the definition below.

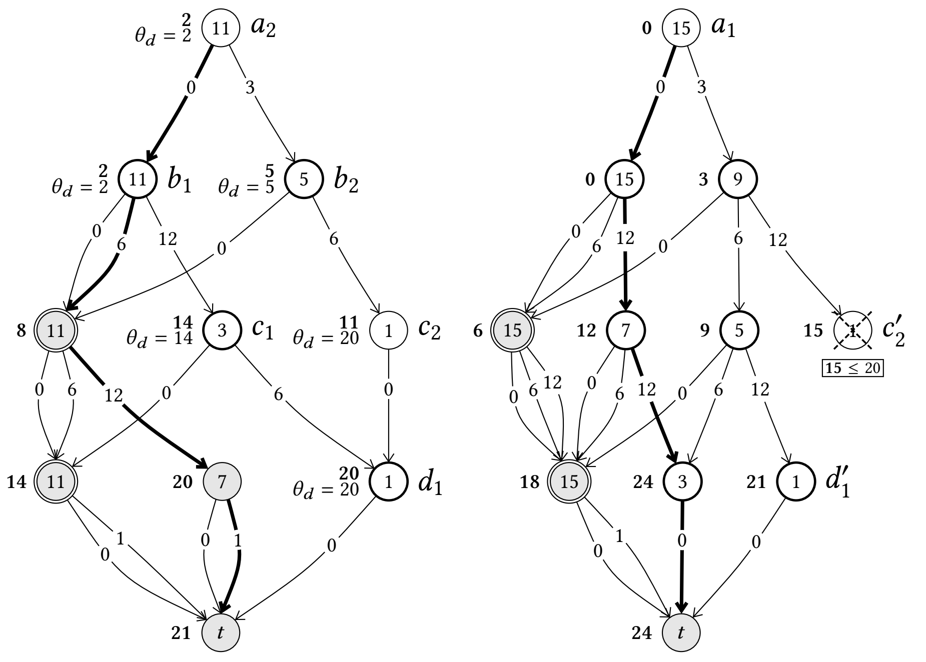
Definition 4.3.2 (Individual dominance threshold). *Given a relaxed DD $\overline{\mathcal{B}}$ and two exact nodes $u_1 \in \overline{\mathcal{B}}$, $u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}})$, the individual dominance threshold of u_1 with respect to u_2 is defined by:*

$$\theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}}) = \theta_{id}(u_2 \mid u_2, \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}), \tag{4.1}$$

$$\theta_{id}(u_2 \mid u_2, \overline{\mathcal{B}}) = v^*(u_2 \mid \overline{\mathcal{B}}). \tag{4.2}$$

Proposition 4.3.1. *Given a relaxed DD $\overline{\mathcal{B}}$, exact nodes $u_1 \in \overline{\mathcal{B}}$, $u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}})$, $u'_1 \notin \overline{\mathcal{B}}$ such that $\sigma(u'_1) = \sigma(u_1)$, and a path $p_1 : r \rightsquigarrow u'_1$, if $v(p_1) > \theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}})$ then $p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) \succ p^*(u_2 \mid \overline{\mathcal{B}})$. Inversely, if $v(p_1) \leq \theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}})$ then $p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) \preceq p^*(u_2 \mid \overline{\mathcal{B}})$.*

Proof. By Definition 4.3.2, we have $v(p_1) > \theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}}) = v^*(u_2 \mid \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$, or equivalently $v(p_1) + v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) > v^*(u_2 \mid \overline{\mathcal{B}})$. Using the path value definition, we obtain $v(p_1) + v(p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) > v(p^*(u_2 \mid \overline{\mathcal{B}}))$. By concatenating the paths p_1 and $p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$, the inequality becomes $v(p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) > v(p^*(u_2 \mid \overline{\mathcal{B}}))$, which by Definition 4.3.1 is equivalent to $p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) \succ p^*(u_2 \mid \overline{\mathcal{B}})$. The proof of the second implication is obtained by replacing each occurrence of $>$ and $\succ$ respectively by $\leq$ and $\preceq$ in the proof above. As $v^*(u_2 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) = 0$, we also trivially have that Equation (4.1) and Equation (4.2) are consistent with each other. □


$\theta_d = \frac{2}{2}$ 11 a_2
$\theta_d = \frac{2}{2}$ 11 b_1
$\theta_d = \frac{5}{5}$ 5 b_2
8 11
$\theta_d = \frac{14}{14}$ 3 c_1
$\theta_d = \frac{11}{20}$ 1 c_2
14 11
20 7
$\theta_d = \frac{20}{20}$ 1 d_1
21 t
0 15 a_1
0 15
3 9
6 15
12 7
9 5
15 1 c_2'
15 ≤ 20
18 15
24 3
21 1 d_1'
24 t

(a) Relaxed DD rooted at a_2 (b) Relaxed DD rooted at a_1

Figure 4.1: Relaxed DDs with $W = 3$ for the FC nodes of Figure 2.3, rooted at (a) a_2 and (b) a_1. Nodes of the relaxed DD (a) are annotated with their dominance threshold, where applicable. The relaxed DD (b) is compiled with respect to the dominance thresholds computed in (a).

Expanding u_1' is required when a $r \rightsquigarrow u_1'$ path satisfies the first condition of Proposition 4.3.1, with respect to any node $u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}})$. Therefore, we define a *dominance threshold* for each exact node $u_1 \in \overline{\mathcal{B}}$ that simply computes the least individual dominance threshold with respect to nodes in $Leaves_d(u_1 \mid \overline{\mathcal{B}})$.

Definition 4.3.3 (Dominance threshold)**.** *Given a relaxed DD $\overline{\mathcal{B}}$, the dominance threshold of an exact node $u_1 \in \overline{\mathcal{B}}$ is given by:*

$$\theta_d(u_1 \mid \overline{\mathcal{B}}) = \begin{cases} \infty, & \text{if } Leaves_d(u_1 \mid \overline{\mathcal{B}}) = \emptyset, \\ \min_{u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}})} \theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}}), & \text{otherwise.} \end{cases}$$

Example 4.3.1. *Let us illustrate the computation of the dominance thresholds with Figure 4.1(a), showing the relaxed DD compiled from node a_2 of our running example. Note that the dominance thresholds can be used alone here because*

RUBs and LocBs are disabled, pruning thresholds would otherwise be needed to obtain correct expansion thresholds. The FC of this relaxed DD consists of the nodes b_1, b_2, c_1 and d_1. By applying Definition 4.3.3, we have for each of them that $\theta_d(u \mid \overline{\mathcal{B}}) = \theta_{id}(u \mid u, \overline{\mathcal{B}}) = v^*(u \mid \overline{\mathcal{B}})$. The dominance thresholds of all other exact nodes can be obtained by bottom-up propagation. For node c_2, we obtain $\theta_d(c_2 \mid \overline{\mathcal{B}}) = \theta_{id}(c_2 \mid d_1, \overline{\mathcal{B}}) = \theta_{id}(d_1 \mid d_1, \overline{\mathcal{B}}) - v(c_2 \to d_1) = 20 - 0 = 20$.

Given those dominance thresholds, Figure 4.1(b) shows the relaxed DD that would be compiled from node a_1. Two pairs of nodes share the same DP state: c_2, c_2' and d_1, d_1'. In (b), node c_2' is pruned since its value is lower than the dominance threshold computed for c_2 in (a), even though c_2' has a greater value than c_2. On the other hand, the dominance threshold computed in (a) for d_1 is not high enough to prevent the expansion of d_1' in (b).

4.3.2 Pruning Thresholds

The second contribution concerns nodes that were pruned either because of their RUB or their LocB, or because of node dominance. It aims at identifying cases where a new path could circumvent a pruning decision that was made during the compilation of a relaxed DD $\overline{\mathcal{B}}$. For that purpose, we define a *pruning threshold* for each node $u_1 \in \overline{\mathcal{B}}$. Similarly to what was done for the dominance thresholds, we denote by $Leaves_p(u_1 \mid \overline{\mathcal{B}})$ the set of successor nodes of u_1 that have been pruned, i.e.

$$Leaves_p(u_1 \mid \overline{\mathcal{B}}) = \left\{ \begin{array}{c} u_2 \in Succ(u_1 \mid \overline{\mathcal{B}}) \mid v^*(u_2 \mid \overline{\mathcal{B}}) + \overline{v}(u_2 \mid \overline{\mathcal{B}}) \leq \underline{v} \text{ or} \\ \exists \Psi' \in Fronts_{layer(u_2)}[\kappa(u_2)] \text{ such that } \Psi' \geq \Psi(u_2) \end{array} \right\}$$

with $\underline{v}$ the value of the incumbent solution at that point. Formally, given a relaxed DD $\overline{\mathcal{B}}$ and three exact nodes $u_1 \in \overline{\mathcal{B}}$, $u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}})$ and $u_1' \notin \overline{\mathcal{B}}$ with $\sigma(u_1') = \sigma(u_1)$, the expansion of node u_1' is required when it is possible that the concatenation $p_1 \cdot p_2$ of a new prefix path $p_1 : r \rightsquigarrow u_1'$ with a suffix path $p_2 : u_1 \rightsquigarrow u_2$ obtains a value sufficient to overcome the past pruning decision about node u_2. This scenario is captured by the *individual pruning threshold* of an exact node $u_1 \in \overline{\mathcal{B}}$ with respect to a node $u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}})$.

Definition 4.3.4 (Individual pruning threshold). *Given a lower bound $\underline{v}$, a relaxed DD $\overline{\mathcal{B}}$, an exact node $u_1 \in \overline{\mathcal{B}}$ and a pruned node $u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}})$,* *the individual pruning threshold of u_1 with respect to u_2 is defined by:*

$$\theta_{ip}(u_1 \mid u_2, \overline{\mathcal{B}}) = \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}), \tag{4.3}$$

$$\theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}}) = \begin{cases} \underline{v} - \overline{v}(u_2 \mid \overline{\mathcal{B}}), & \text{if } v^*(u_2 \mid \overline{\mathcal{B}}) + \overline{v}(u_2 \mid \overline{\mathcal{B}}) \leq \underline{v}, \\ v' - 1, & \text{if } \exists \Psi' \in Fronts_{layer(u_2)}[\kappa(u_2)], \\ & \quad \text{with } \Psi' = (v') \cdot \psi', \text{ such that} \\ & \quad \Psi' \geq \Psi(u_2) \text{ and } \psi' = \psi(u_2), \\ v', & \text{if } \exists \Psi' \in Fronts_{layer(u_2)}[\kappa(u_2)], \\ & \quad \text{with } \Psi' = (v') \cdot \psi', \text{ such that} \\ & \quad \Psi' \geq \Psi(u_2) \text{ and } \psi' \neq \psi(u_2). \end{cases} \tag{4.4}$$

Proposition 4.3.2. *Given a lower bound $\underline{v}$, a relaxed DD $\overline{\mathcal{B}}$, a pruned node $u \in \overline{\mathcal{B}}$ and a node $u' \notin \overline{\mathcal{B}}$ such that $\sigma(u') = \sigma(u)$, and a path $p : r \rightsquigarrow u'$. If $v(p) > \theta_{ip}(u \mid u, \overline{\mathcal{B}})$, then either:*

- *$v(p) + \overline{v}(u \mid \overline{\mathcal{B}}) > \underline{v}$, in case node u was pruned by its RUB or LocB,*
- *or, in case it was pruned by a utility $\Psi' \in Fronts_{layer(u)}[\kappa(u)]$ such that $\Psi' \geq \Psi(u)$, then $\Psi' \ngeq \Psi(u')$.*

Inversely, if $v(p) \leq \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$, then either:

- *$v(p) + \overline{v}(u \mid \overline{\mathcal{B}}) \leq \underline{v}$, in case node u was pruned by its RUB or LocB,*
- *or, in case it was pruned by a utility $\Psi' \in Fronts_{layer(u)}[\kappa(u)]$ such that $\Psi' \geq \Psi(u)$, then $\Psi' \geq \Psi(u')$.*

Proof. Let us prove the first implication in the case where node u was pruned by its RUB or LocB. If we have that $v(p) > \theta_{ip}(u \mid u, \overline{\mathcal{B}})$, then by applying Equation (4.4), we obtain $v(p) > \underline{v} - \overline{v}(u \mid \overline{\mathcal{B}})$ or equivalently $v(p) + \overline{v}(u \mid \overline{\mathcal{B}}) > \underline{v}$. In case node u was pruned by a utility $\Psi' = (v') \cdot \psi'$ split as v' its value and ψ' its partial utility, then $v(p) > \theta_{ip}(u \mid u, \overline{\mathcal{B}})$ can be expanded using Equation (4.4) as

$$v(p) > \begin{cases} v' - 1, & \text{if } \psi' = \psi(u), \\ v', & \text{otherwise.} \end{cases}$$

If $\psi' = \psi(u)$, then $v(p) > v' - 1$, or evenly $v(p) \geq v'$, and thus $\Psi' \ngeq \Psi(u')$. On the other hand, if $\psi' \neq \psi(u)$, then $v(p) > v'$ and clearly also $\Psi' \ngeq \Psi(u')$. The proof of the second implication is obtained by replacing each occurrence of $>$ by $\leq$, of $\geq$ by $<$ and of $\ngeq$ by $\geq$ in the proof above. □

Proposition 4.3.3. *Given a lower bound $\underline{v}$, a relaxed DD $\overline{\mathcal{B}}$, an exact node $u_1 \in \overline{\mathcal{B}}$, a pruned node $u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}})$, an exact node $u_1' \notin \overline{\mathcal{B}}$ such* *that* $\sigma(u_1') = \sigma(u_1)$, *and a path* $p_1 : r \rightsquigarrow u_1'$. *If* $v(p_1) > \theta_{ip}(u_1 \mid u_2, \overline{\mathcal{B}})$ *then* $v(p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) > \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$. *Inversely, if* $v(p_1) \leq \theta_{ip}(u_1 \mid u_2, \overline{\mathcal{B}})$ *then* $v(p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) \leq \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$.

Proof. By Equation (4.3), we have $v(p_1) > \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$, or equivalently $v(p_1) + v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) > \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$. Using the path value definition, we obtain $v(p_1) + v(p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) > \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$. By concatenating the paths p_1 and $p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$, the inequality becomes $v(p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) > \theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$. The proof of the second implication is obtained by replacing each occurrence of $>$ by $\leq$ in the proof above. As $v^*(u_2 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) = 0$, we also trivially have that Equation (4.3) and Equation (4.4) are consistent with each other. □

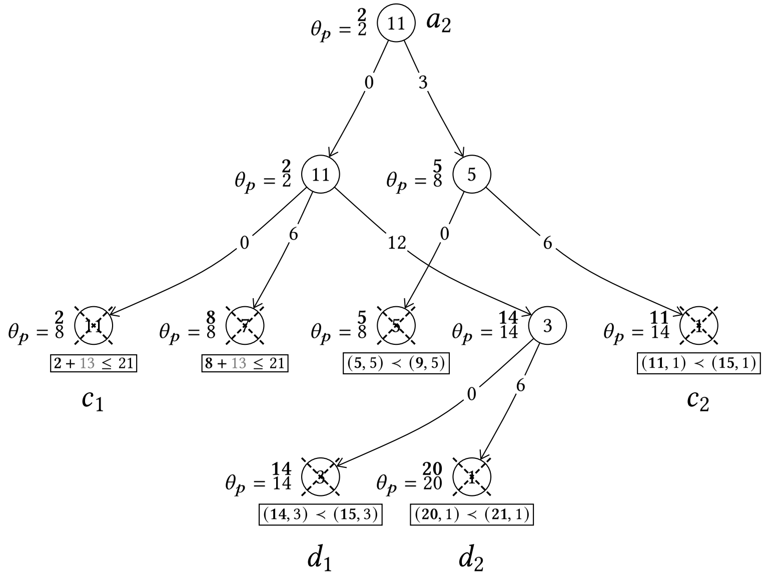
As for the dominance threshold, finding a $r \rightsquigarrow u_1'$ path satisfying the first condition of Proposition 4.3.3 with respect to any node $u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}})$ is a sufficient condition to require the expansion of node u_1'. The *pruning threshold* of each exact node $u_1 \in \overline{\mathcal{B}}$ is thus computed as the least individual pruning threshold with respect to any node in $Leaves_p(u_1 \mid \overline{\mathcal{B}})$.

Definition 4.3.5 (Pruning threshold). *Given a lower bound* $\underline{v}$, *a relaxed DD* $\overline{\mathcal{B}}$ *and an exact node* $u_1 \in \overline{\mathcal{B}}$, *the pruning threshold of* u_1 *within* $\overline{\mathcal{B}}$ *is given by:*

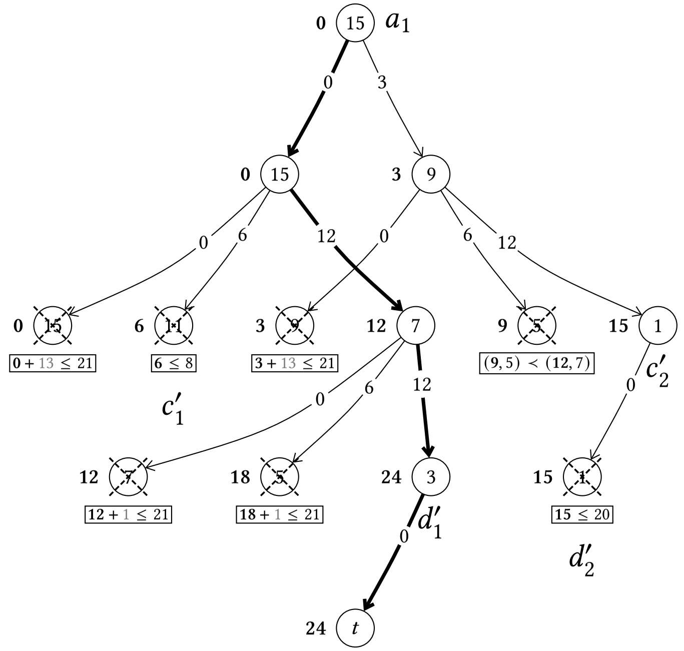
$$\theta_p(u_1 \mid \overline{\mathcal{B}}) = \begin{cases} \infty, & \text{if } Leaves_p(u_1 \mid \overline{\mathcal{B}}) = \emptyset, \\ \min_{u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}})} \theta_{ip}(u_1 \mid u_2, \overline{\mathcal{B}}), & \text{otherwise.} \end{cases}$$

Example 4.3.2. *We consider the same scenario as in Example 4.3.1 of a relaxed DD rooted at node* a_2*, but this time with RUBs and LocBs enabled. As shown by Figure 4.2(a), the RUBs and the node dominance relations manage to prune every path of the DD before reaching the terminal node. Therefore, we can showcase the pruning thresholds alone, otherwise we should combine them with the dominance thresholds, as will be explained in the next section. For each node pruned by the RUB, we have by Definition 4.3.5 that* $\theta_p(u \mid \overline{\mathcal{B}}) = \theta_{ip}(u \mid u, \overline{\mathcal{B}}) = \underline{v} - \overline{v}(u \mid \overline{\mathcal{B}})$. *For instance, we have* $\theta_p(c_1 \mid \overline{\mathcal{B}}) = 21 - 13 = 8$. *For nodes pruned by a utility* $\Psi' = (v') \cdot \psi'$, *we have for each of them that* $\psi(u) = \psi'$. *Therefore, we apply the second case of Equation* (4.4), *i.e.* $\theta_p(u \mid \overline{\mathcal{B}}) = v' - 1$. *For node* c_2, *we obtain* $\theta_p(c_2) = 15 - 1 = 14$ *since it is dominated by* $\Psi' = (15, 1)$.

Once these pruning thresholds are obtained, they can impact the compilation of a relaxed DD from node a_1*, as illustrated by Figure 4.2(b). Among the nodes that share a common DP state across DDs (a) and (b),* c_1' *and* d_2' *are successfully filtered by the pruning thresholds computed for* c_1 *and* d_2*. Note that, like in Example 4.3.2, the value obtained for* c_1' *in (b) is greater than the one of* c_1 *in (a). The expansion of nodes* c_2' *and* d_1' *cannot be avoided. However, a new incumbent*



(a) Relaxed DD rooted at a_2



(b) Relaxed DD rooted at a_1

Figure 4.2: Relaxed DDs with $W = 3$ for the FC nodes of Figure 2.3, rooted at (a) a_2 and (b) a_1. Nodes of the relaxed DD (a) are annotated with their pruning threshold. The relaxed DD (b) is compiled with respect to the pruning thresholds computed in (a).

solution is found after expanding node d_1': $\underline{x} = (0, 0, 2, 2, 0)$ with $\underline{v} = f(\underline{x}) = 24$, which is the optimal solution of the problem. This concludes the resolution of our BKP instance by the B&B algorithm since both diagrams of Figure 4.2 are exact and a_1 and a_2 were the only two nodes in the EC of Figure 2.3.

4.3.3 Expansion Thresholds

Sections 4.3.1 and 4.3.2 presented the two scenarios that necessitate the expansion of nodes associated with a DP state previously reached by an exact node in a relaxed DD, along with the dominance and pruning thresholds used to detect them. This section explains how these two thresholds are combined to form an *expansion threshold*, and provides a proof of correctness of the accompanying pruning criterion. It also details the algorithm for computing the expansion threshold for all exact nodes in a relaxed DD.

Dominance and pruning thresholds each give a sufficient condition for the expansion of a node with a previously reached DP state. The definition of the expansion threshold given below simply ensures that the expansion is triggered if any of those two conditions are met, and inversely that it is avoided if none of those two conditions are met.

Definition 4.3.6 (Expansion threshold). *Given a relaxed DD $\overline{\mathcal{B}}$, the expansion threshold of an exact node $u_1 \in \overline{\mathcal{B}}$ is defined by:*

$$\theta(u_1 \mid \overline{\mathcal{B}}) = \min \left\{ \theta_d(u_1 \mid \overline{\mathcal{B}}), \theta_p(u_1 \mid \overline{\mathcal{B}}) \right\}.$$

Proposition 4.3.4. *Given a relaxed DD $\overline{\mathcal{B}}$, an exact node $u_1 \in \overline{\mathcal{B}}$, an exact node $u_1' \notin \overline{\mathcal{B}}$ such that $\sigma(u_1') = \sigma(u_1)$, and a path $p_1 : r \rightsquigarrow u_1'$, if $v(p_1) \leq \theta(u_1 \mid \overline{\mathcal{B}})$ then p_1 can be pruned.*

Proof. If $v(p_1) \leq \theta(u_1 \mid \overline{\mathcal{B}})$ then by definition of the expansion threshold, we have $v(p_1) \leq \theta_d(u_1 \mid \overline{\mathcal{B}})$ and $v(p_1) \leq \theta_p(u_1 \mid \overline{\mathcal{B}})$. For the dominance threshold inequality, we have by Definition 4.3.3 that $\forall u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}}) : v(p_1) \leq \theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}})$. When applying Proposition 4.3.1, we obtain $\forall u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}}) : v(p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) \leq p^*(u_2 \mid \overline{\mathcal{B}})$. The second inequality, concerning the pruning threshold, can be developed similarly: by Definition 4.3.5, $\forall u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}}) : v(p_1) \leq \theta_{ip}(u_1 \mid u_2, \overline{\mathcal{B}})$. By substituting with Proposition 4.3.3, $\forall u_2 \in Leaves_p(u_1 \mid \overline{\mathcal{B}}) : v(p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})) \leq \theta_{ip}(u_2 \mid u_1, \overline{\mathcal{B}})$. In that case, Proposition 4.3.2 guarantees that either:

- $v(p) + \overline{v}(u \mid \overline{\mathcal{B}}) \leq \underline{v}$, in case node u was pruned by its RUB or LocB,
- or, in case it was pruned by a utility $\Psi' \in Fronts_{layer(u)}[\kappa(u)]$ such that $\Psi' \geq \Psi(u)$, that $\Psi' \geq \Psi(u')$.

Therefore, if $v(p_1) \leq \theta(u_1 \mid \overline{\mathcal{B}})$ then the concatenation of p_1 with any path $u_1 \rightsquigarrow u_2$ with $u_2 \in Leaves_d(u_1 \mid \overline{\mathcal{B}}) \cup Leaves_p(u_1 \mid \overline{\mathcal{B}})$ is guaranteed to be either weakly dominated or pruned. Furthermore, by definition of $Leaves_d(u_1 \mid \overline{\mathcal{B}})$ and $Leaves_p(u_1 \mid \overline{\mathcal{B}})$, their union covers all successors of u_1 that are in $EC(\overline{\mathcal{B}})$ or that are pruned. It thus constitutes an exhaustive decomposition of the corresponding subproblem, concluding our argument that p_1 is irrelevant for the search. □

In Definitions 4.3.2 and 4.3.4, one can notice that the individual dominance and pruning thresholds $\theta_{id}(u_1 \mid u_2, \overline{\mathcal{B}})$ and $\theta_{ip}(u_1 \mid u_2, \overline{\mathcal{B}})$ only depend on the value $v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$, apart from the thresholds $\theta_{id}(u_2 \mid u_2, \overline{\mathcal{B}})$ and $\theta_{ip}(u_2 \mid u_2, \overline{\mathcal{B}})$. For a relaxed DD rooted at node u_r, if those initial individual thresholds are correctly set for nodes in $Leaves_d(u_r \mid \overline{\mathcal{B}})$ and $Leaves_p(u_r \mid \overline{\mathcal{B}})$, the expansion thresholds can thus be computed by performing a single bottom-up pass on the relaxed DD $\overline{\mathcal{B}}$, as described by Algorithm 8.

The thresholds are first initialized at line 3 with a default value of ∞ for all nodes. Then, the loops at lines 4 and 5 iterate through all nodes of the DD, starting with the nodes of the last layer up to those of the first layer, and propagate threshold values in a bottom-up fashion. Depending on whether they are pruned or belong to the EC, several nodes require a careful initialization of their expansion threshold.

- **Pruned by the *Cache*:** line 6 checks whether the *Cache* already contains an expansion threshold greater than $v^*(u \mid \overline{\mathcal{B}})$. If so, the node was pruned and the previously computed threshold can simply be recycled, as it remains valid throughout the entire algorithm.

- **Pruned by the *Fronts* or the RUB:** when reaching a node u pruned by a node dominance relation at line 9, or with its RUB during the top-down compilation at line 11, both lines 10 and 12 set its expansion threshold to $\theta_{ip}(u \mid u, \overline{\mathcal{B}})$ the individual pruning threshold of u relative to itself, because it has no other successor.

- **Pruned by LocB:** if a node u of the EC was pruned because of its LocB however, the expansion threshold is computed at line 15 as the minimum between $\theta_{ip}(u \mid u, \overline{\mathcal{B}})$ the individual pruning threshold relative to u and the current value of $\theta(u \mid \overline{\mathcal{B}})$, which accounts for the individual dominance and pruning thresholds relative to successors of u.

- **EC node:** the expansion threshold of each cutset node u that will be added to the *Fringe* is simply given by $\theta_{id}(u \mid u, \overline{\mathcal{B}})$ the individual dominance threshold of u relative to itself, as shown at line 17.

Algorithm 8 Computation of the threshold $\theta(u \mid \overline{\mathcal{B}})$ of every exact node u in a relaxed DD $\overline{\mathcal{B}}$ and update of the *Cache*.

```
 1: $i \leftarrow$ index of the root layer of $\overline{\mathcal{B}}$
 2: $(L_i, \dots, L_n) \leftarrow Layers(\overline{\mathcal{B}})$
 3: $\theta(u \mid \overline{\mathcal{B}}) \leftarrow \infty$ **for all** $u \in \overline{\mathcal{B}}$
 4: **for** $j = n$ **down to** $i$ **do**
 5:     **for all** $u \in L_j$ **do**
 6:         **if** $Cache.contains(\sigma(u))$ **and** $v^*(u \mid \overline{\mathcal{B}}) \leq \theta(Cache.get(\sigma(u)))$ **then**
 7:             $\theta(u \mid \overline{\mathcal{B}}) \leftarrow \theta(Cache.get(\sigma(u)))$
 8:         **else**
 9:             **if** $\exists \Psi' \in Fronts_j[\kappa(u)]$ s.t. $\Psi' \geq \Psi(u)$ **then**        // dom. pruning
10:                 $\theta(u \mid \overline{\mathcal{B}}) \leftarrow v' - 1$ **if** $\psi' = \psi(u)$ **else** $v'$ (with $\Psi' = (v') \cdot \psi'$)
11:             **else if** $v^*(u \mid \overline{\mathcal{B}}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}$ **then**            // RUB pruning
12:                 $\theta(u \mid \overline{\mathcal{B}}) \leftarrow \underline{v} - \overline{v}_{rub}(\sigma(u))$
13:             **else if** $u \in EC(\overline{\mathcal{B}})$ **then**
14:                 **if** $v^*(u \mid \overline{\mathcal{B}}) + \overline{v}_{locb}(u \mid \overline{\mathcal{B}}) \leq \underline{v}$ **then**          // LocB pruning
15:                     $\theta(u \mid \overline{\mathcal{B}}) \leftarrow \min\left\{\theta(u \mid \overline{\mathcal{B}}), \underline{v} - \overline{v}_{locb}(u \mid \overline{\mathcal{B}})\right\}$
16:                 **else**
17:                     $\theta(u \mid \overline{\mathcal{B}}) \leftarrow v^*(u \mid \overline{\mathcal{B}})$
18:             **if** $u$ is exact **then**
19:                 $expanded \leftarrow$ false **if** $u \in EC(\overline{\mathcal{B}})$ **else** true
20:                 $Cache.insertOrReplace(\sigma(u), \langle\theta(u \mid \overline{\mathcal{B}}), expanded\rangle)$
21:         **for all** arc $a = (u' \xrightarrow{d} u)$ incident to $u$ **do**
22:             $\theta(u' \mid \overline{\mathcal{B}}) \leftarrow \min\left\{\theta(u' \mid \overline{\mathcal{B}}), \theta(u \mid \overline{\mathcal{B}}) - v(a)\right\}$
```

Lines 21 and 22 then take care of the bottom-up propagation of the expansion thresholds as to obtain the correct threshold values for all nodes. The last step at line 20 is to save the expansion threshold computed for each exact node in the *Cache* for later use. The *Cache* is an associative array storing *key-value* pairs $\langle\sigma(u), \theta(u \mid \overline{\mathcal{B}})\rangle$, implemented as a *hash table* in practice. Note that Algorithm 8 can be applied to any type of EC, but an important detail is that thresholds should not be inserted in the *Cache* for exact successors of nodes in the EC to allow for their later expansion. As the FC contains all the deepest exact nodes, the algorithm can be applied as is. For the LEL however, a check must be added at line 18.

In addition to the threshold value, a flag called *expanded* is stored in the *Cache* to distinguish two types of expansion thresholds. First, the nodes that have been added to the *Fringe* from the EC of a relaxed DD $\overline{\mathcal{B}}$ with $\theta(u \mid \overline{\mathcal{B}}) = v^*(u \mid \overline{\mathcal{B}})$ and that need to be further explored by the B&B, therefore

Algorithm 9 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:     $pruned \leftarrow \emptyset$
5:     perform dominance pruning using Algorithm 7
6:     perform cached-based pruning using Algorithm 10
7:     $L'_j \leftarrow L_j \setminus pruned$
8:     if $|L'_j| > W$ then
9:         restrict or relax the layer to get $W$ nodes with Algorithm 2
10:    $L_{j+1} \leftarrow \emptyset$
11:    for all $u \in L'_j$ do
12:        if $v^*(u \mid \mathcal{B}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}$ then        // RUB pruning
13:            continue
14:        for all $d \in D_j$ do
15:            create node $u'$ with state $\sigma(u') = t_j(\sigma(u), d)$
               or retrieve it from $L_{j+1}$
16:            create arc $a = (u \xrightarrow{d} u')$ with $v(a) = h_j(\sigma(u), d)$ and $l(a) = d$
17:            add $u'$ to $L_{j+1}$ and add $a$ to $A$
18: merge nodes in $L_n$ into terminal node $t$
```

Algorithm 10 Cache-based filtering of layer L_j of a DD $\mathcal{B}$.

```
1: if $j > i$ then        // pruning for the root done in Algorithm 11
2:     for all $u \in L_j$ do
3:         if $Cache.contains(\sigma(u))$ and $v^*(u \mid \mathcal{B}) \leq \theta(Cache.get(\sigma(u)))$ then
4:             $pruned \leftarrow pruned \cup \{u\}$
```

being associated with an expanded flag set to *false* at line 20 of Algorithm 8. And second, all exact nodes above the EC of any relaxed DD $\overline{\mathcal{B}}$ in which the threshold $\theta(u \mid \overline{\mathcal{B}})$ was obtained. In this case, they can be considered as expanded since all their outgoing paths were either pruned or cross a node in the EC of $\overline{\mathcal{B}}$. When those thresholds are inserted in the *Cache* at line 20 of Algorithm 8, their *expanded* flag is thus set to *true*.

4.3.4 Filtering the Search Using the Cache

Now that we have presented how expansion thresholds can be computed for all exact nodes of relaxed DDs, we can now elaborate on how these values are utilized within the B&B. There are two places where the *Cache* might be beneficial and prune some nodes. As mentioned before, and shown in Examples 4.3.1 and 4.3.2, the first is in the top-down compilation of approximate

Algorithm 11 The DD-based branch-and-bound algorithm with caching.

```
1: $Fringe \leftarrow \{r\}$   // a priority queue ordered by decreasing $v(u) + \overline{v}(u)$
2: $Fronts_j \leftarrow \emptyset$ for $j = 0, \dots, n$ // hash tables of dom. keys to Pareto fronts
3: $Cache \leftarrow \emptyset$   // a hash table of states to threshold $\sigma(u) \rightarrow \langle \theta, expanded \rangle$
4: $\underline{x} \leftarrow \bot, \underline{v} \leftarrow -\infty$   // incumbent solution and its value
5: while $Fringe$ is not empty do
6:     $u \leftarrow$ best node from $Fringe$, remove it from $Fringe$
7:     if $v(u) + \overline{v}(u) \leq \underline{v}$ then
8:         continue
9:     if $Cache.contains(\sigma(u))$ then
10:        $\langle \theta, expanded \rangle \leftarrow Cache.get(\sigma(u))$
11:        if $v(u) < \theta$ or $(v(u) = \theta \wedge expanded)$ then
12:            continue
13:    $\underline{\mathcal{B}} \leftarrow Restricted(u)$   // compile restricted DD with Algorithm 9
14:    if $v^*(\underline{\mathcal{B}}) > \underline{v}$ then   // update incumbent
15:        $\underline{x} \leftarrow x^*(\underline{\mathcal{B}}), \underline{v} \leftarrow v^*(\underline{\mathcal{B}})$
16:    if $\underline{\mathcal{B}}$ is not exact then
17:        $\overline{\mathcal{B}} \leftarrow Relaxed(u)$   // compile relaxed DD with Algorithm 9
18:        compute LocBs with Algorithm 3 applied to $\overline{\mathcal{B}}$
19:        update $Cache$ with Algorithm 8 applied to $\overline{\mathcal{B}}$
20:        for all $u' \in EC(\overline{\mathcal{B}})$ do
21:            $v(u') \leftarrow v^*(u' \mid \overline{\mathcal{B}}), \overline{v}(u') \leftarrow \overline{v}(u' \mid \overline{\mathcal{B}}), p(u') \leftarrow p^*(u' \mid \overline{\mathcal{B}})$
22:            if $v(u') + \overline{v}(u') > \underline{v}$ then
23:                add $u'$ to $Fringe$
24: return $(\underline{x}, \underline{v})$
```

DDs, as described by Algorithm 9. As for the dominance-based filtering introduced in Chapter 3, the cache-based filtering is specified in a separate procedure given by Algorithm 10, and that collects the filtered nodes in the *pruned* set. At line 3 of Algorithm 10, the *Cache* is queried and whenever the value $v^*(u \mid \mathcal{B})$ of a node u fails to surpass the threshold θ, it is added to the *pruned* set. Nodes from the *pruned* set are kept in the original layer L_j so that the threshold computations of Algorithm 8 can be performed correctly.

Furthermore, the *Cache* can be used when selecting a node for exploration in the B&B loop. At line 9 of Algorithm 11, the *Cache* is queried to potentially retrieve an entry with a threshold value as well as an *expanded* flag. If a threshold is indeed present, two cases arise depending on the value of the *expanded* flag. If it is *True*, the node is ignored if its value is less or equal to the threshold. When *False* however, the node is only ignored if its value is strictly less than the threshold, because in case of equality, this node will be the first to be expanded with that value. As this filtering happens at the B&B level, it is not repeated for the root node of approximate DDs, see line 1 of Algorithm 9.

An interesting implication of the use of expansion thresholds is that during the bottom-up traversal of relaxed DDs done to compute LocBs, it is possible to detect *dangling nodes*, i.e. nodes with no feasible outgoing transition. As they have no path to the terminal node, their LocB is $-\infty$ and the expansion threshold stored is ∞. As a result, the bottom-up propagation of the thresholds permits to detect both suboptimality and infeasibility earlier in the future compilations of approximate DDs, in addition to dominance relations discussed before. In terms of complexity, computing the thresholds requires a single traversal of the relaxed DDs compiled, so it has the same complexity as Algorithm 9 and Algorithm 3 that respectively handle the top-down compilation and the computation of the LocBs.

4.4 Limitations

The thresholds presented in Section 4.3 offer new pruning perspectives that will be shown to be very impactful in Section 4.6. Yet, they exhibit two main limitations that are discussed in this section.

4.4.1 Memory Consumption

In order to apply B&B with caching, extra information must be stored in memory. Some effort is done to reduce memory consumption by deleting thresholds as soon as they are no longer required by the algorithm. A sufficient condition to remove a threshold is when it concerns a node located in a layer above the *first active layer.*

Definition 4.4.1 (First active layer)**.** *Given $\mathcal{B}$ the exact DD for problem $\mathcal{P}$ with layers $L_0, \ldots, L_n$. The first active layer of the B&B specified by Algorithm 11 is defined as the least index j such that a node of layer L_j is in the Fringe or is currently selected for exploration at line 6.*

Thresholds related to nodes above the first active layer can be safely removed from the *Cache* as there is no way of reaching the associated DP states again. If memory consumption was nevertheless an issue, a simple solution would be to delete an arbitrary subset of the thresholds stored in the *Cache*. This does not compromise the optimality guarantees of the algorithm since the only effect of thresholds is to prevent the expansion of nodes associated with already visited DP states. However, removing some thresholds decreases the pruning perspectives of the algorithm. Some memory will therefore be saved at the cost of speed. One could even imagine using eviction rules based on an activity measure of the thresholds in the *Cache*. Note that all the algorithms presented in the chapter are written as if the *Cache* could remove some thresholds along the way.

An argument that could be held against the use of the *Cache* in the context of DD-based B&B is that in the worst case, it still might need to accommodate as many nodes as there are nodes in the exact DD encoding the problem being solved. While this argument is true, we would like to point out that the aforementioned explosive worst-case memory requirement already plagued the original B&B algorithm as it does not implement any measures to limit the size of the *Fringe*. Furthermore, we would like to emphasize that maintaining the *Cache* is no more costly than maintaining the *Fringe* and significantly less expensive than memorizing an actual instantiation of the exact DD. Indeed, the maintenance of both the *Cache* and the *Fringe* is $O(|U|)$ – where U is the set of nodes in the state space – whereas the requirement to store an actual instantiation of the exact DD in memory is $O(|U| + |A|)$ where A is the set of arcs connecting the nodes in U. It is also worth mentioning that in that case, $|A|$ dominates $|U|$ since by definition of the domains and transition relations, the size of A is bounded by the product of the domain sizes $\Pi_{j=0,\dots,n-1}|D_j|$.

4.4.2 Variables Orderings

Many discrete optimization problems have an imposed variable ordering in their natural DP model. For instance, DP models for sequencing problems usually make decisions about the j-th position of the sequence at the j-th stage of the DP model [CH13]. For DP models that allow it, however, it has been shown that variable orderings can yield exact DDs of reduced size as well as approximate DDs with tighter bounds [Ber+12; Cap+22; KH22]. When DDs are used within a B&B algorithm, variable orderings thus constitute an additional heuristic that can speed up the search. They can be separated in two categories: *static* and *dynamic* variable orderings. The former refers to a single variable ordering used for all DD compilations during the B&B. The latter denotes a heuristic that dynamically decides which variable to branch on when generating the next layer of a DD, based on the states contained in the current layer. Since the techniques introduced in this chapter are solely based on the overlapping structure of the DP models, a dynamic variable ordering would most likely compromise much of the expected pruning. Indeed, it seems unlikely that many states would overlap in DDs compiled with different variable orderings, although it ultimately depends on the modeling of each specific problem.

4.5 Applications

All the problems used in the experimental study of Chapter 3 are also part of the experiments of this chapter, using the same benchmark instances and parameters. There are no additional modeling ingredients to specify for those problems to apply the caching strategy described in this chapter. This section provides the complete modeling of two additional problems, and for which no obvious dominance rule exists.

4.5.1 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: $Q^i_p \in \{0, 1\}$ 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.

To give a better understanding of the problem, we hereby recall the MIP model denoted PIG-A-1 in [PW06]. Variables $x^i_p \in \{0, 1\}$ decide whether an item of type i is produced at time period p. On the other hand, variables $y^i_p \in \{0, 1\}$ decide whether the machine is ready to produce an item of type i at time period p. Indeed, the machine can be idle at certain periods. Variables $q^i_p \in \mathbb{N}_0$ accumulate the quantity of items of type i stored at period p. Finally, variables $\chi^{i,j}_p \in \{0, 1\}$ capture a changeover between item types i and j at period p.

$$\min \sum_{i \in I} \sum_{p=0}^{H-1} S_i q^i_p + \sum_{i,j \in I} \sum_{p=0}^{H-1} C_{ij} \chi^{i,j}_p \tag{4.5}$$

$$q^i_{p-1} + x^i_p = q^i_p + Q^i_p \qquad \forall i \in I, 0 \leq p < H \tag{4.6}$$

$$q^i_{-1} = 0 \qquad \forall i \in I \tag{4.7}$$

$$x^i_p \leq y^i_p \qquad \forall i \in I, 0 \leq p < H \tag{4.8}$$

$$\sum_{i \in I} y^i_p = 1 \qquad \forall 0 \leq p < H \tag{4.9}$$

$$\chi^{i,j}_p \geq y^i_{p-1} + y^i_p - 1 \qquad \forall i, j \in I, 0 < p < H \tag{4.10}$$

Equation (4.6) models the stocking of the items and consumes them when needed, with the quantities initialized by Equation (4.7). Then, Equation (4.8) ensures that the machine is in the correct mode to produce an item and Equation (4.9) allows only one mode for each time period. Finally, Equation (4.10) sets the correct value for the changeover variables.

The PSP was already tackled with a DD-based approach in [GS22; Gil22]. We hereby recall the same formulation, except for a few changes that allow to solve scenarios that require the machine to be idle for some time periods.

4.5.1.1 Dynamic Programming Formulation

In this DP 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 definition of 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 $\langle i, R\rangle$ where i is the item type produced at the next time period – and thus scheduled at the previous transition – and R is a vector where R_i gives the remaining number of demands to satisfy for item type i.

- Control variables: $x_j \in N'$ with $0 \leq j < H$ decides the item type to produce at period j.
- State space: $\mathcal{S} = \{s \mid s.i \in N', \forall i \in N, 0 \leq s.R_i \leq \sum_{0 \leq p < H} Q^i_p\}$. The root state is given by $\hat{r} = \langle \perp, (\sum_{0 \leq p < H} Q^0_p, \dots, \sum_{0 \leq p < H} Q^{n-1}_p)\rangle$ and the terminal states are of the form $\langle i, (0, \dots, 0)\rangle$ with $i \in N'$.
- Transition functions:

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

where

$$t^i_j(s^j, x_j) = \begin{cases} x_j, & \text{if } x_j \neq \perp \\ s^j.i, & \text{otherwise.} \end{cases}$$

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

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: $v_r = 0$.

4.5.1.2 Relaxation

The merging operator is defined as follows:

$$\oplus(\mathcal{M}) = \left\langle \perp, (\min_{s \in \mathcal{M}} s.R_0, \ldots, \min_{s \in \mathcal{M}} s.R_{n-1}) \right\rangle.$$

As merged states might disagree on the item type produced before, the configuration of the machine is always reset to the dummy item type $\perp$. For each item type, the minimum remaining number of items to produce is computed, meaning that any demand satisfied by a least one state is considered satisfied in the merged state. As for the TSPTW, the relaxed transition value operator is the identity function $\Gamma_{\mathcal{M}}(v, u) = v$.

4.5.1.3 Rough Lower Bound

When the changeover costs are ignored, the PSP falls under the Wagner-Whitin conditions [PW06] that allow to compute the optimal stocking cost for a given set of remaining items to produce. Conversely, if the stocking costs and the delivery constraints are omitted, the PSP can be reduced to the TSP. Therefore, a valid lower bound on the total changeover cost to produce a remaining set of items is to take the total weight of a Minimum Spanning Tree computed on the graph of changeover costs limited to item types that still need to be produced. The optimal weight for all these spanning trees can be precomputed because the number of items is usually small. As there is no overlap between the two lower bounds described, the RLB for the PSP can sum their individual contributions to obtain a stronger lower bound.

4.5.1.4 Experimental Setting

A set of randomly generated instances was created with the number of items in $n \in \{6, 8, 10\}$, the number of periods in $H \in \{100, 150, 200\}$ and the density in $\{0.9, 0.95, 1\}$. The density is computed as the number of demands over the number of time periods. To generate instances with diverse proportions between the stocking costs and the changeover costs, we created pairs of bounds $\rho \in \{(100, 100000), (1000, 10000), (10000, 1000), (100000, 100)\}$ and for each ρ, the stocking costs were sampled in the interval $[0, \rho_1]$ and the pairwise changeover costs in the interval $[0, \rho_2]$. Additionally, a parameter $K \in \{6, 8, 10\}$ controls the number of groups of item types having similar characteristics, and stocking and changeover costs are sampled around a same value for item types of the same group, similarly to what was done for the separation matrices of ALP instances, as described in Section 3.4.2.

To generate the demands, item type and time period pairs were selected uniformly among all possible values and added to the instance as long as it remained feasible. Several fixed widths were used in the experiments following the formula $W = \alpha \times n$, where n is the number of variables of each particular instance and $\alpha \in \{1, 10, 100\}$ a multiplying factor.

4.5.2 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.

4.5.2.1 Dynamic Programming Formulation

A DP model for TalentSched was introduced in [GSC11] that we slightly adapt here to make it suitable for the state merging-based relaxation. 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 space: $\mathcal{S} = \{(M, P) \mid M, P \subseteq N, M \cap P = \emptyset\}$. The root state is $\hat{r} = (N, \emptyset)$ 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: $v_r = 0$.

4.5.2.2 Relaxation

Sets of scenes that must and can still be scheduled are merged exactly like the location sets for the TSPTW. The merging operator is thus defined as

$$\oplus(\mathcal{M}) = (\oplus_M(\mathcal{M}), \oplus_P(\mathcal{M}))$$

where

$$\oplus_M(\mathcal{M}) = \bigcap_{s \in \mathcal{M}} s.M$$
$$\oplus_P(\mathcal{M}) = (\bigcup_{s \in \mathcal{M}} s.M \cup s.P) \setminus (\bigcap_{s \in \mathcal{M}} s.M).$$

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.

4.5.2.3 Rough Lower Bound

We use the complex lower bound given by Theorem 1 in [GSC11] as an RLB for our approach, which we do not develop here for conciseness.

4.5.2.4 Experimental Setting

We generated a set of 400 random instances with different number of scenes $n \in \{22, 24, 26, 28\}$ and actors $m \in \{10, 15\}$. A parameter $\rho \in \{0.3, 0.4\}$ controlling the average fraction of actors required for each scene was also used to generate instances with diverse densities of actors. The scenes were grouped in $K \in \{15, 20, 25\}$ groups among which similar actor requirements are generated. A fixed width of $W = 100$ is used for all experiments concerning the TalentSched.

4.6 Computational Experiments

This section presents the results of the computational experiments that were conducted to evaluate the impact of the additional pruning techniques presented in this chapter. The proposed caching mechanism was coupled to the two configurations studied in Chapter 3 – DDO and DDO+D – and thus gave two new configurations DDO+C and DDO+D+C. Again, each solver and configuration was allotted 600 seconds for each benchmark instance of every optimization problem discussed. The results presented for DDO and DDO+D on the TSPTW, the ALP, the LCS and the KP are the same as those analyzed in Chapter 3. For all the experiments, the best results were obtained by using FCs when enabling the *Cache*, and otherwise using LELs. It is worth mentioning that the baseline approach already uses duplicate state detection in the *Fringe*. For the PSP, the DD-based approach is compared with the MIP model referred to as PIG-A-3 in [PW06], solved with Gurobi 9.5.2 [Gur22].

4.6.1 Impact of the Caching Mechanism

Figures 4.3 to 4.6 show the results of these experiments, respectively in terms of computation time and number of DD nodes expanded during the search. Each graph presents the total number of instances solved under any given time or nodes expanded. For each of the studied problems, DDO+C is able to solve significantly more instances than DDO within the time limit. For the TSPTW and the ALP, DDO+C is almost as performant as DDO+D without any knowledge of their problem-specific dominance rules. Concerning the LCS, DDO+C only solves 9 more instances than DDO, whereas DDO+D could solve 26 instances that DDO could not. For the KP, however, DDO+C largely outperforms both DDO and DDO+D. These improvements in terms of solving time are directly linked to a reduction of the number of nodes expanded by the algorithm, as can be seen in Figures 4.5 and 4.6. This confirms our intuition that a lot of work is unnecessarily repeated by DDO and shows that the pruning techniques introduced in this chapter help neutralize much of this

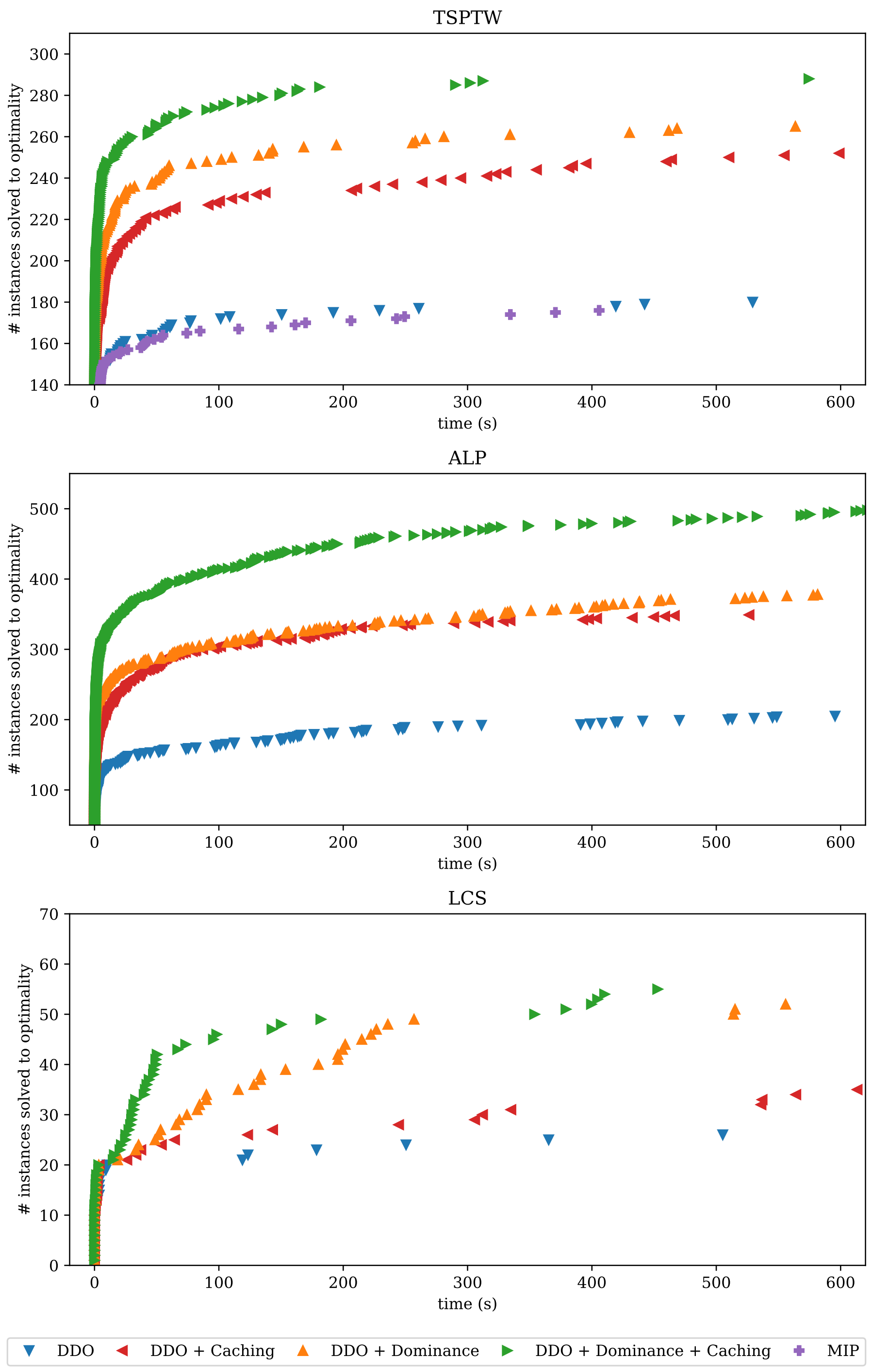


Figure 4.3: Number of instances solved over time by DDO, DDO+C, DDO+D, DDO+D+C and Gurobi for three different problems.

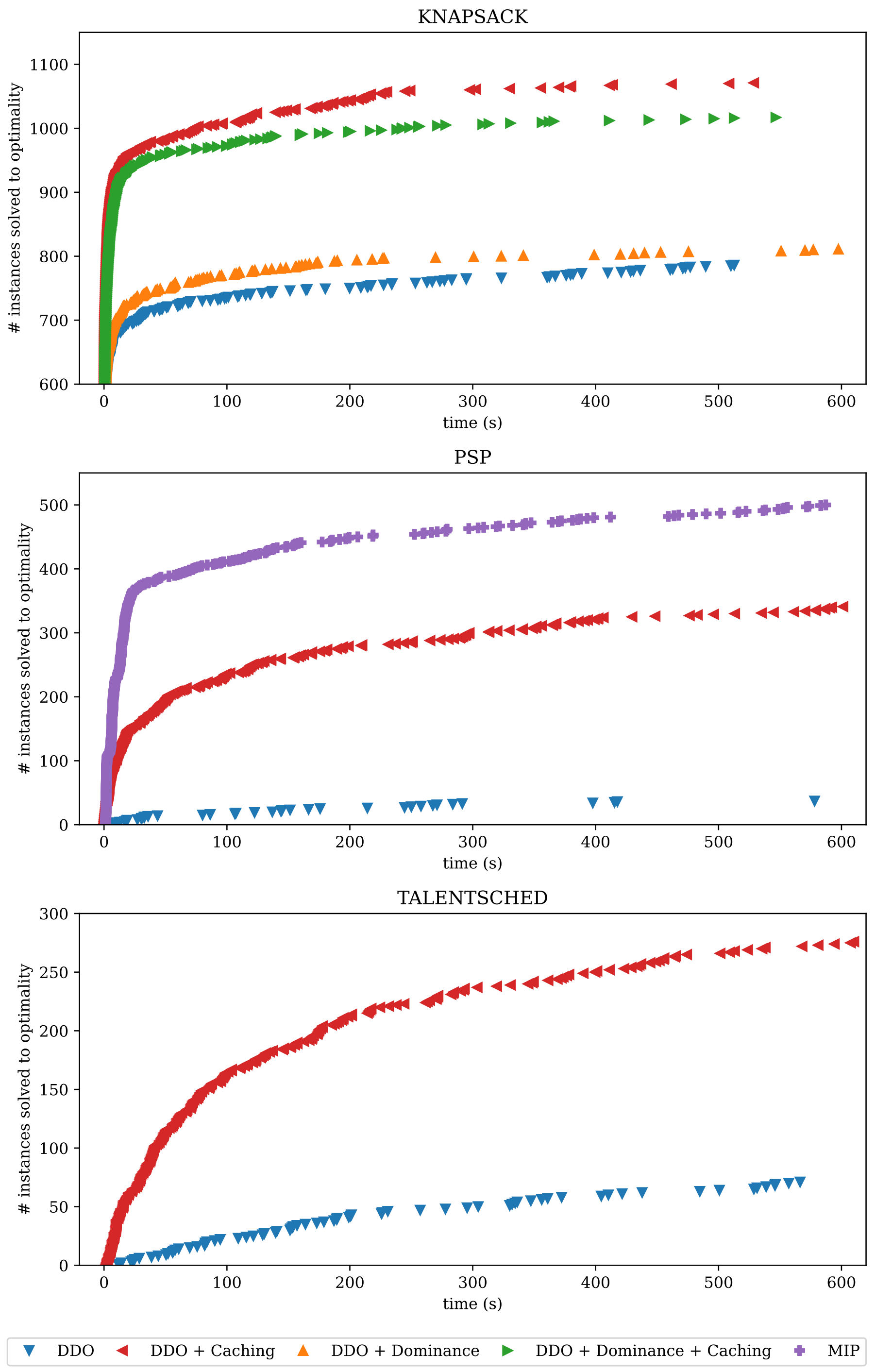


Figure 4.4: Number of instances solved over time by DDO, DDO+C, DDO+D, DDO+D+C and Gurobi for three different problems.

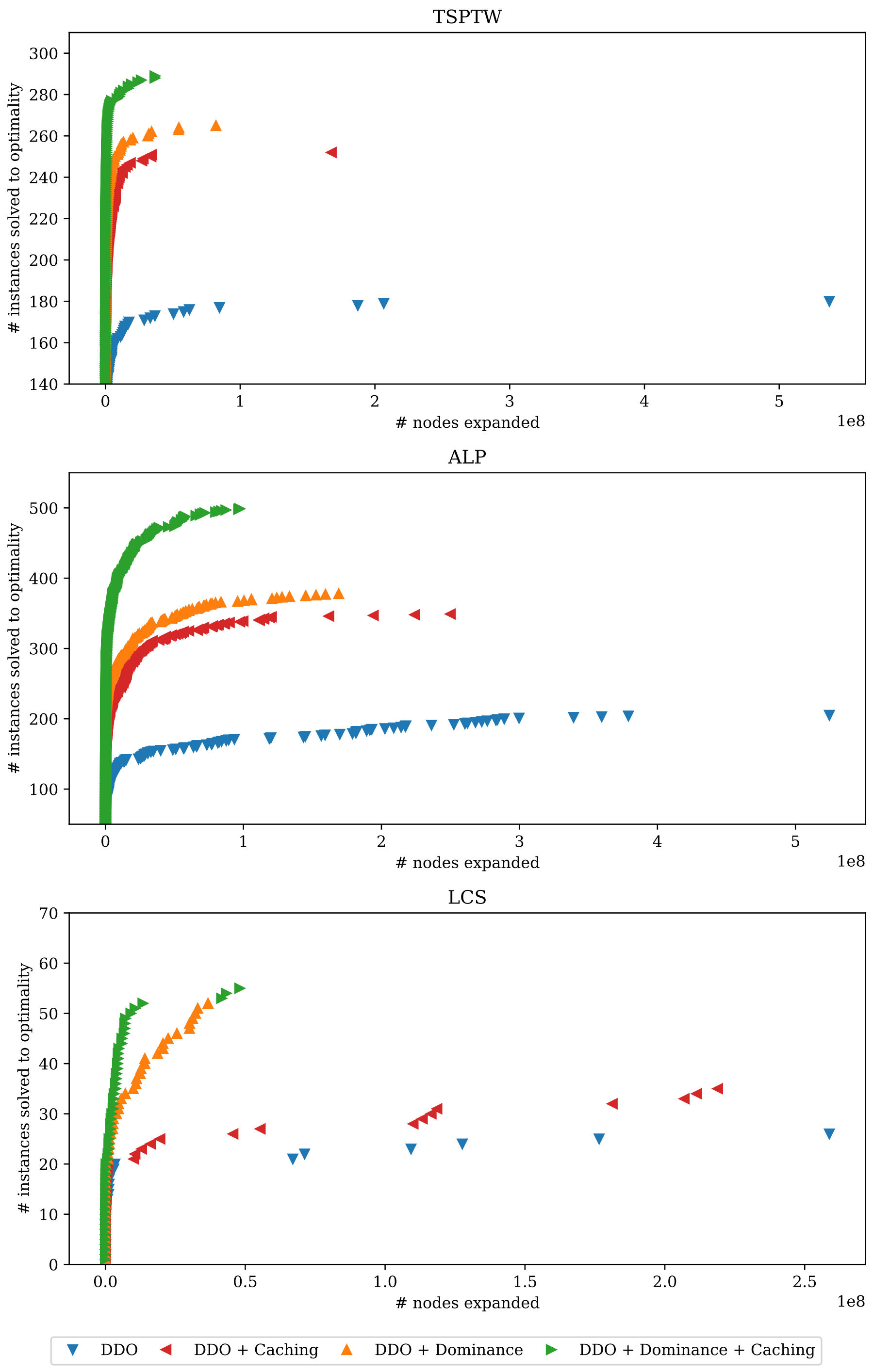


Figure 4.5: Number of instances solved by DDO, DDO+C, DDO+D and DDO+D+C with respect to the number of DD nodes expanded for three different problems.

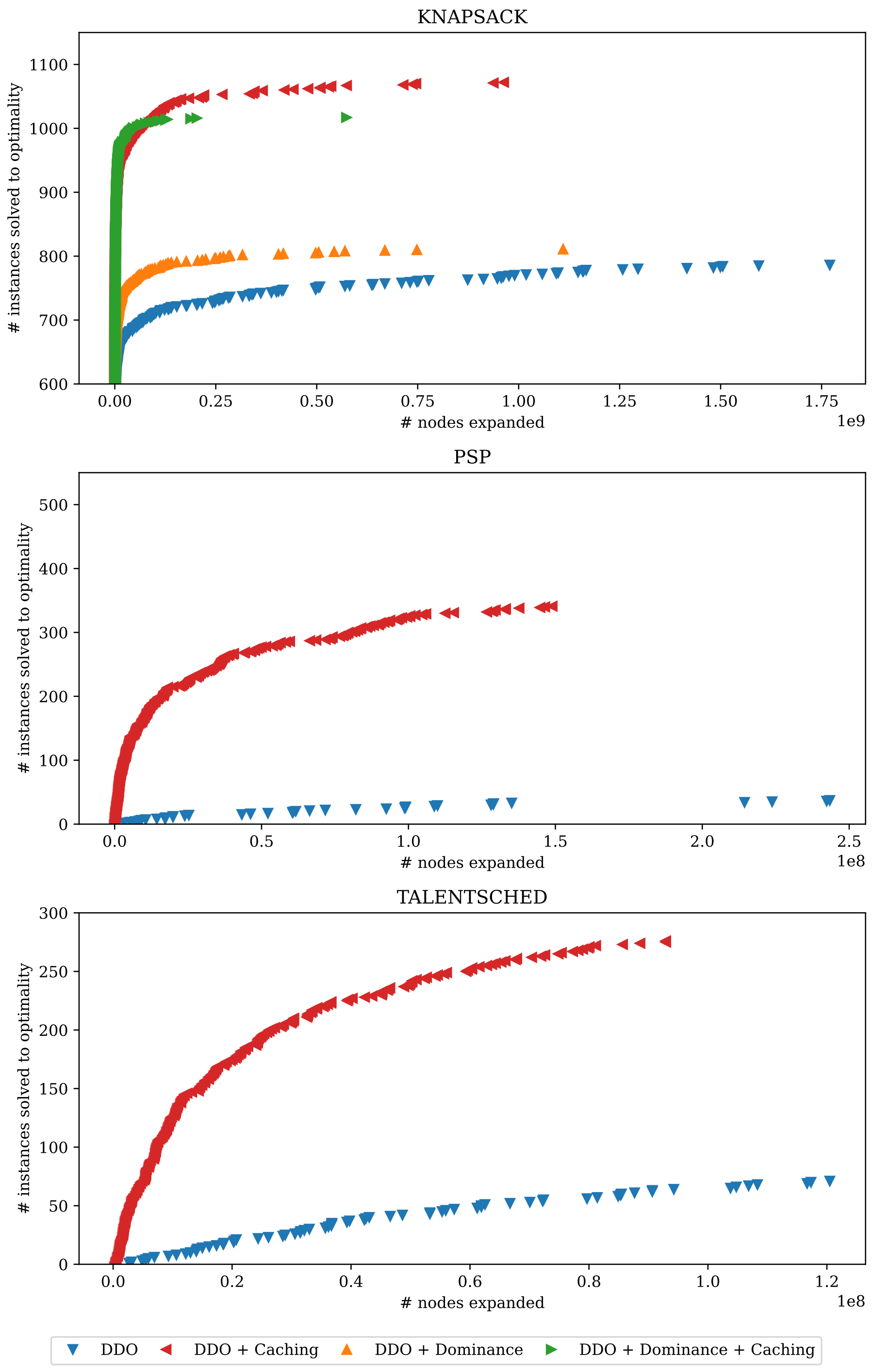


Figure 4.6: Number of instances solved by DDO, DDO+C, DDO+D and DDO+D+C with respect to the number of DD nodes expanded for three different problems.

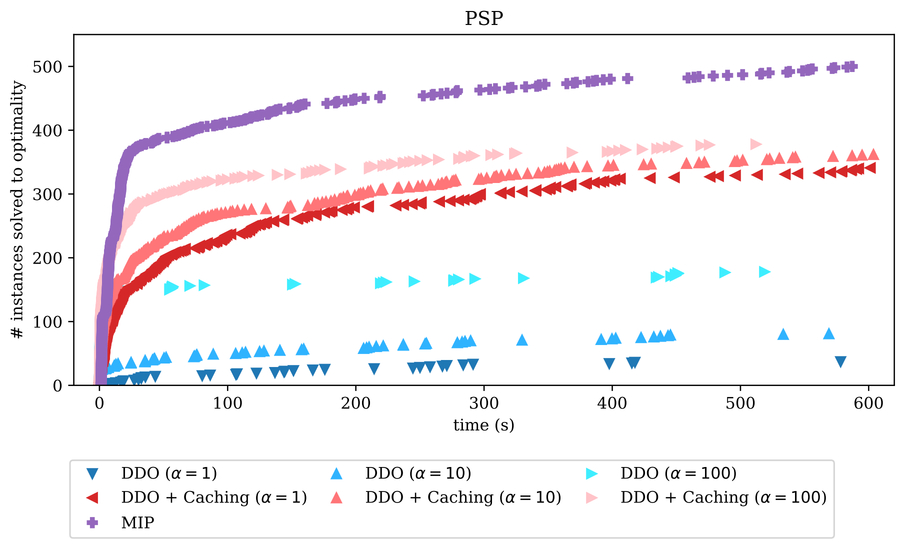


Figure 4.7: Number of instances solved over time by DDO, DDO+C, DDO+D and DDO+D+C for three different problems.

problem. The expansion thresholds allow discarding many transitions during the compilation of approximate DDs and avoid repeating previous work.

Moreover, if we look at the performance achieved using different maximum widths for the PSP on Figure 4.7, one can notice that for DDO, increasing the width helps to solve many more instances: with $\alpha = 10$ and $\alpha = 100$, DDO respectively closes 44 and 141 additional instances, compared to DDO with $\alpha = 1$. Indeed, larger DDs allow stronger bounds to be derived and instances to be closed more quickly, as long as they are not too expensive to compile. Yet, this disparity is much less significant for DDO+C: with $\alpha = 10$ and $\alpha = 100$, DDO+C respectively closes 21 and 37 additional instances, compared to DDO+C with $\alpha = 1$. This perfectly captures the double benefit of the *Cache*. The strength of dominance and pruning thresholds allow discarding many transitions during the compilation of approximate DDs and avoid repeating previous work. As a result, narrower DDs can explore and prune the search space almost as fast while being much cheaper to generate.

Another benefit of the *Cache* is that it fully exploits the potential of FCs: as all non-improving transitions are blocked by the *Cache*, it is only natural to use the deepest possible exact cutset. The same cannot be said in the case of DDO because FCs usually contain many nodes, some of which are the parents of others, and this only exacerbates the caveats mentioned in Section 4.2.

To put these results into perspective, we compare them on Figures 4.3 and 4.7 to those obtained with MIP models solved by Gurobi. For the TSPTW, Gurobi could only solve 176 instances, which is less than the worst-performing configuration of DDO. On the other hand, it could solve 500 PSP instances while DDO+C with $\alpha = 100$ solved 378 of them, and DDO performed much worse, solving only 178 instances with $\alpha = 100$. Therefore, even if the best results are achieved with the MIP model, using the *Cache* closed much of the gap that separates the two techniques.

4.6.2 Synergy with the Dominance Rules

Quite remarkably, Figures 4.3 and 4.5 show that combining the dominance rules with the caching strategy results in even better performance for the TSPTW, the ALP and the LCS, compared to those two improvements used separately. This is a good indication that the bottom-up propagation of the expansion thresholds introduced in this chapter is able to deduce dominance and suboptimality pruning criteria that cannot be derived simply through problem-specific dominance rules and RUBs. Inversely, problem-specific dominance rules allow filtering nodes a priori based on a dominance relation existing between two nodes reached during the search, for which expansion expansions would possibly never find a connection. For the KP, we can observe on Figures 4.4 and 4.6 that DDO+D+C performs better than DDO+D but worse than DDO+C. This gives further evidence of the limitations described in Section 3.5 about the dominance rule for the KP.

For the TSPTW and the ALP, we again analyze the quality of the first solution found by DDO+D and DDO+D+C, and the iteration at which the solution is found – see Figure 4.8. Indeed, for the other problems where a solution is guaranteed to be found with the root restricted DD, the *Cache* has no influence. For the TSPTW, the first solution found by either configuration is almost always the same, only in 7 cases DDO+D finds a better solution than DDO+D+C and in 4 cases in the other direction. However, we can see that this solution is found later in the search by DDO+D+C in 18 cases, whereas the inverse occurs only 4 times. This phenomenon can happen when compiling approximate DDs from a given node, because the expansion thresholds associated with nodes inserted in the *Fringe* block their exploration unless a better value is obtained. Therefore, each given restricted DD is confined to a smaller part of the search space that might remove or make it harder to find some good solutions, that will eventually be discovered when starting the compilation from another node. Yet, for the ALP, we can see that DDO+D+C finds a feasible solution for 12 more instances that DDO+D. This can be attributed to the node reduction brought by the *Cache*, which allows exploring the search space faster and thus sometimes reach parts of the search space that DDO+D could not. The quality of the solutions and the iteration at which they are found is otherwise similar for both configurations.

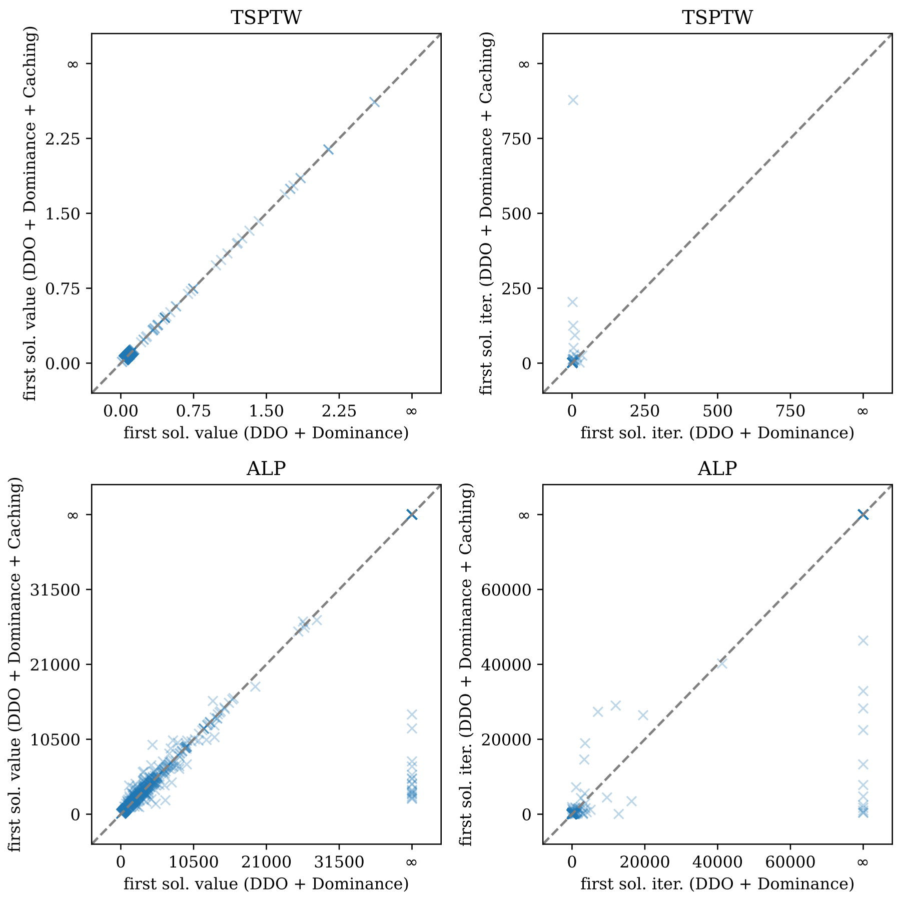


Figure 4.8: Comparison of the value of the first solution obtained for each instance by DDO+D and DDO+D+C, and of the iteration at which this solution is found.

4.6.3 Memory Consumption

The performance improvements discussed in Sections 4.6.1 and 4.6.2 do not come completely for free, as the *Cache* must store all the thresholds computed during the B&B algorithm. It is thus important to study the impact of this technique on the memory consumption of the algorithm. For every instance solved by each algorithm, the peak amount of memory used during the execution was recorded. Figures 4.9 and 4.10 show the number of instances solved using a given maximum amount of memory. We can clearly see an increase in memory consumption when enabling the *Cache* for most of the instances solved by all configurations. However, when the peak amount of memory exceeds around 100MB, DDO+C starts solving more instances of all

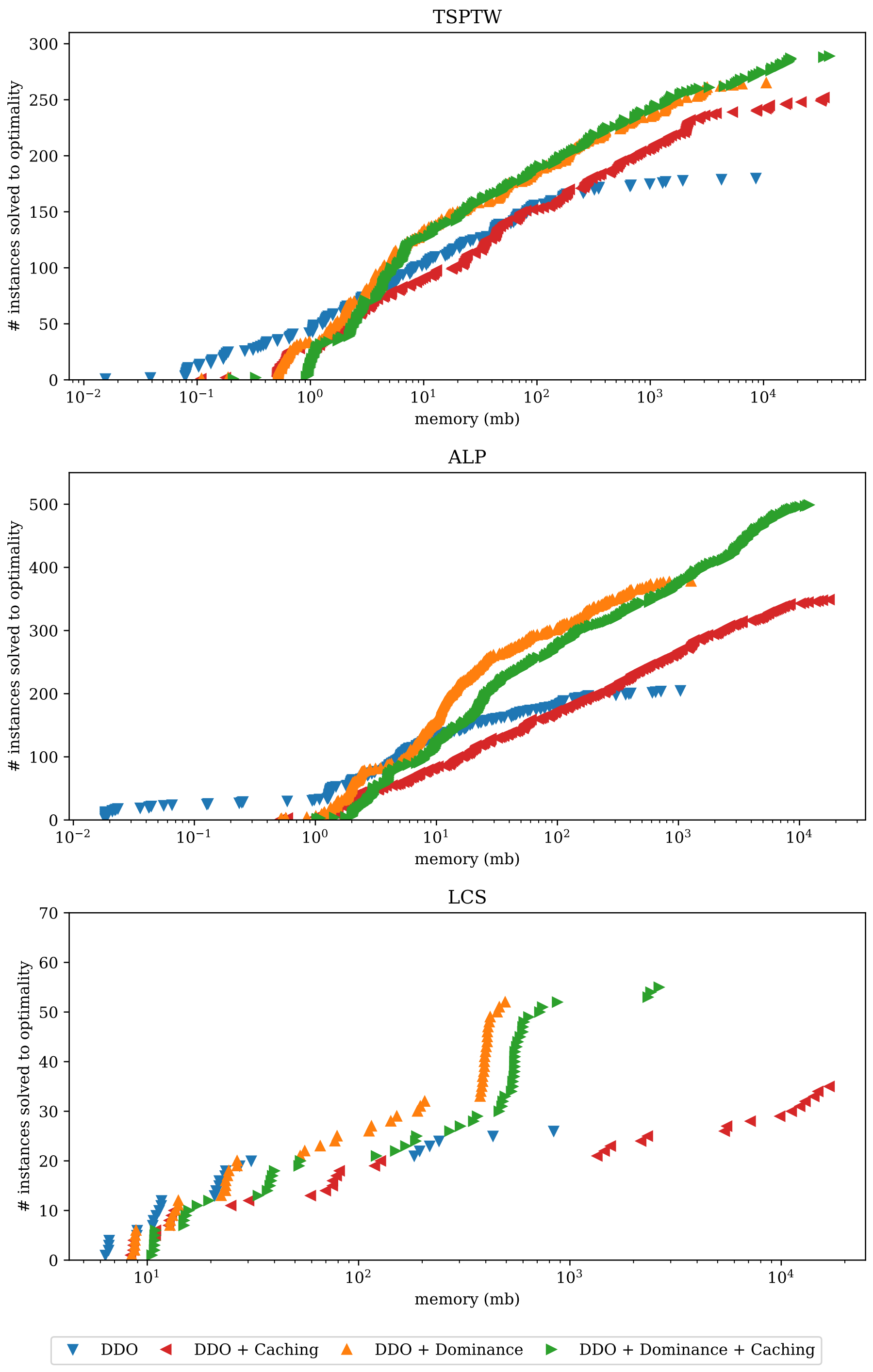


Figure 4.9: Number of instances solved by DDO, DDO+C, DDO+D and DDO+D+C for three different problems, with respect to the peak amount of memory used.

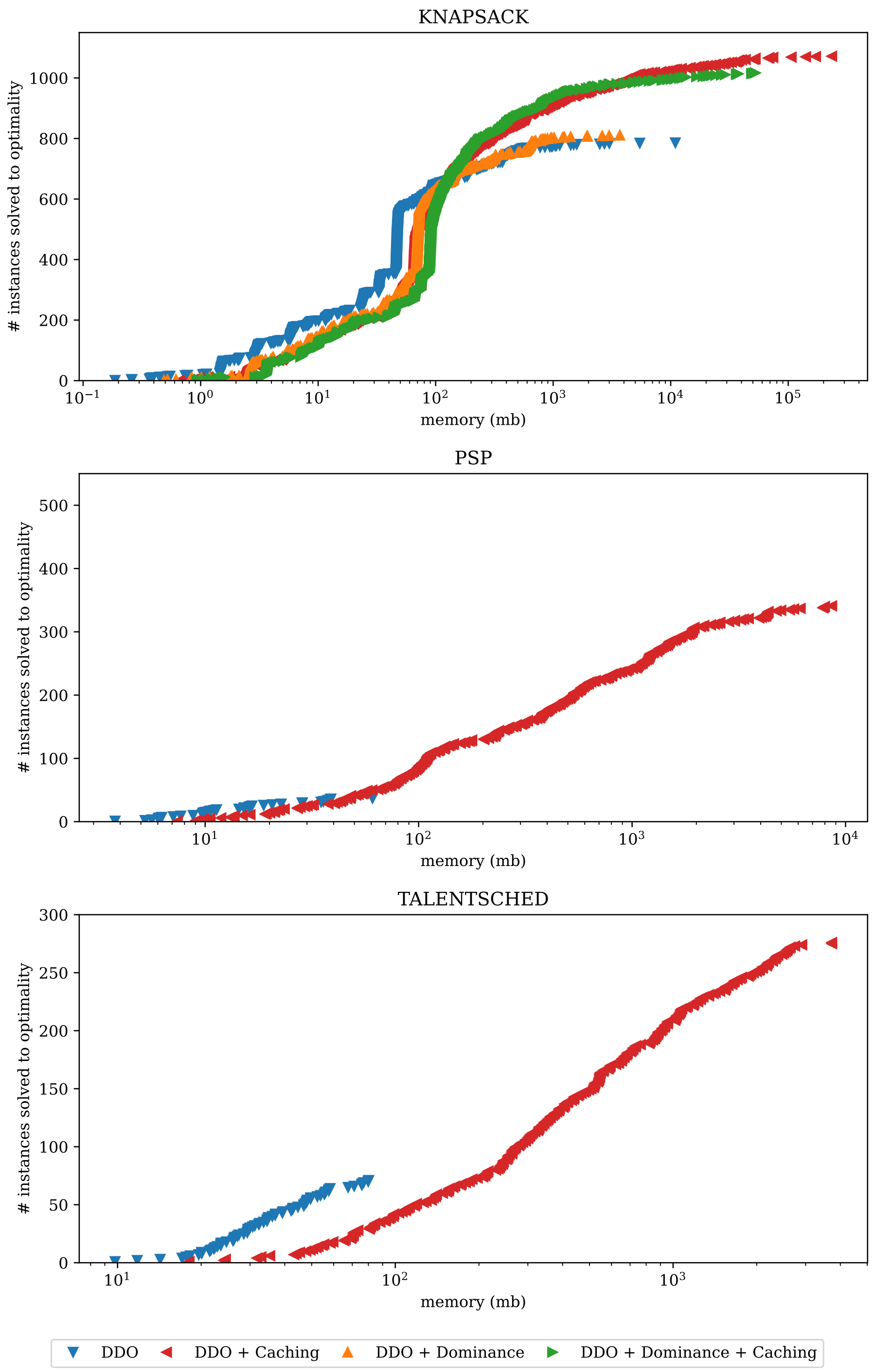


Figure 4.10: Number of instances solved by DDO, DDO+C, DDO+D and DDO+D+C for three different problems, with respect to the peak amount of memory used.

problems with a same given amount of memory and under the given time limit. Thus, even in terms of memory consumption, the cost of maintaining the *Cache* seems to be compensated by its pruning effect which causes DDs to be sparser and the *Fringe* smaller. Interestingly, for problems with dominance rules, DDO+D+C shows a lower memory consumption than DDO+C, while also maintaining the *Fronts*. Indeed, the dominance rules manage to filter nodes a priori during the compilation of approximate DDs, and thus sometimes before even computing and storing a threshold for them.

4.7 Conclusion

In this chapter, we first discussed how the DD-based B&B algorithm tends to repeatedly explore overlapping parts of the search space. Then, we introduced dominance and pruning thresholds with the intention of overcoming this limitation. The former propagate dominance relations between partial solutions obtained within approximate DDs while the latter allow the approach to be combined and strengthened by the pruning performed by LocBs, RUBs and dominance rules. Both types of thresholds are used in a single caching and pruning mechanism that is able to discard many nodes associated with previously visited DP states during subsequent DD compilations. Finally, we presented experimental results that clearly show the impact of the techniques introduced: B&B with caching vastly outperforms the classical B&B algorithm in terms of instances solved on the six discrete optimization problems studied in the chapter. Furthermore, the combination of the expansion thresholds with the dominance checking procedures described in Chapter 3 yields even better results than either ingredient used in isolation. Finally, the experiments showed that the memory consumption induced by the *Cache* was either acceptable or even overcompensated by its pruning effect for difficult instances, as well as mitigated when used in conjunction of dominance rules.

In general, we expect that using the *Cache* will be beneficial for solving problems whenever the theoretical search tree of the DP model comprises isomorphic subtrees, which are normally superimposed in the corresponding exact DD. That is, we expect that the *Cache* to be profitable whenever pure DP would be an efficient – albeit impractical because of memory limitations – method for solving the problem at hand.

Aggregate Dynamic Programming-Based Bounds and Heuristics

5

This chapter is largely based on the following paper: V. Coppé, X. Gillard, and P. Schaus. "Boosting Decision Diagram-Based Branch-And-Bound by Pre-Solving with Aggregate Dynamic Programming". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023. It translates ideas from aggregate dynamic programming to the DD-based optimization framework, and incorporates them as bounds and heuristics that guide the search.

5.1 Introduction

As discussed in the previous chapters, optimizations techniques based on DP and DDs can prove highly effective. In some cases, however, the size and structure of the DP state spaces is prohibitive for the bounds derived from restricted and relaxed DDs to be tight. 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 chapter presents a framework for integrating *aggregate dynamic programming* [Axs83; BBS87] ideas with DD-based optimization that aims to address some of these shortcomings. 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 an upper 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 upper 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.

All the components of this aggregation-disaggregation framework are introduced in Section 5.2 and illustrated by providing two different aggregation schemes for the BKP. In Section 5.3, we then explain how the B&B algorithm can be modified to replace the state merging-based relaxation by the aggregation-based one, and thus avoid compiling relaxed DDs. Section 5.4 then applies the framework to the TalentSched, PSP and ALP. These problems are then used in Section 5.5 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.

5.2 Aggregate Dynamic Programming for Decision Diagrams

This section introduces all the components needed to integrate aggregate dynamic programming with DD-based optimizations. An illustration of the framework is given by Figure 5.1 where we distinguish two phases. First, the *pre-solving* phase that transforms a given instance in an aggregate and relaxed version, as explained in Section 5.2.1, and solves it by compiling the associated exact DD $\mathcal{B}'$. Then, the *exploitation* phase refers to the process of solving the original instance and capitalizing on the precomputed information by deriving upper bounds and node selection heuristics that speed up the

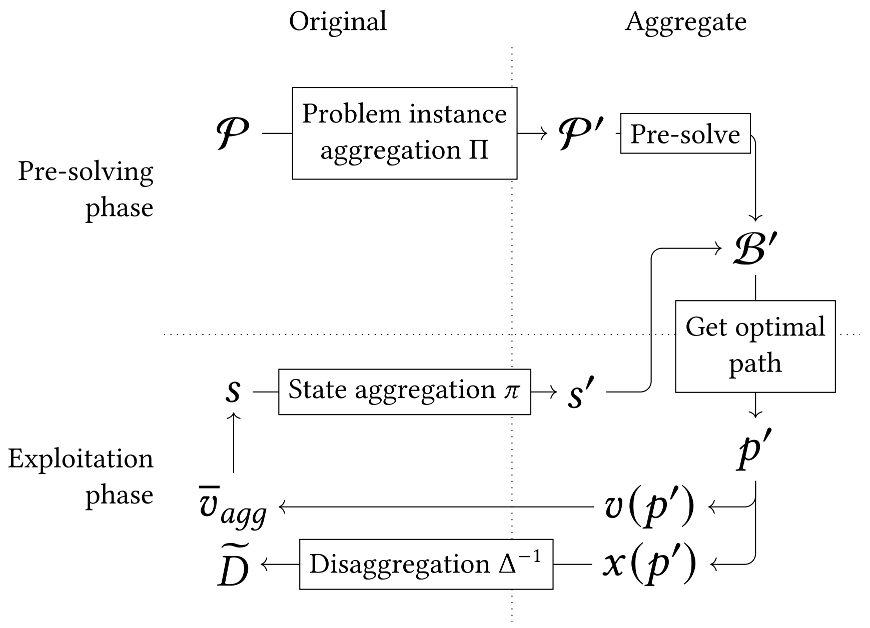


Figure 5.1: Illustration of the aggregation-disaggregation framework, with $\overline{v}_{agg}$ and $\widetilde{D}$ respectively denoting the aggregation-based upper bound and node selection heuristic.

search. Those two means of guiding the search are presented in Sections 5.2.2 and 5.2.3 respectively.

5.2.1 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. An *instance aggregation operator* Π is defined such that the aggregate problem instance $\mathcal{P}' = \Pi(\mathcal{P})$ is a relaxation of the original problem instance $\mathcal{P}$. The aggregate problem instance $\mathcal{P}'$ is associated with a set of n' aggregate variables $x'_0, \ldots, x'_{n'-1}$ taking values in an aggregate domain $\mathcal{D}' = \mathcal{D}'_0 \times \cdots \times \mathcal{D}'_{n'-1}$, and with an aggregate objective function f'. Formally, if we dispose of a *solution aggregation operator* $\Delta : \mathcal{D} \longrightarrow \mathcal{D}'$ that maps each assignment of the original problem to its aggregate equivalent, Π yields a relaxation if and only if for all $x \in Sol(\mathcal{P})$ we have that $\Delta(x) \in Sol(\mathcal{P}')$ and $f(x) \leq f'(\Delta(x))$.

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,

V	W	Q
2	4	1
3	4	1
6	4	2
6	2	2
1	4	1
$C = 15$		

Table 5.1: The aggregate BKP instance for aggregation (A).

V	W	Q
3	4	2
6	2	4
1	5	1
$C = 15$		

Table 5.2: The aggregate BKP instance for aggregation (B).

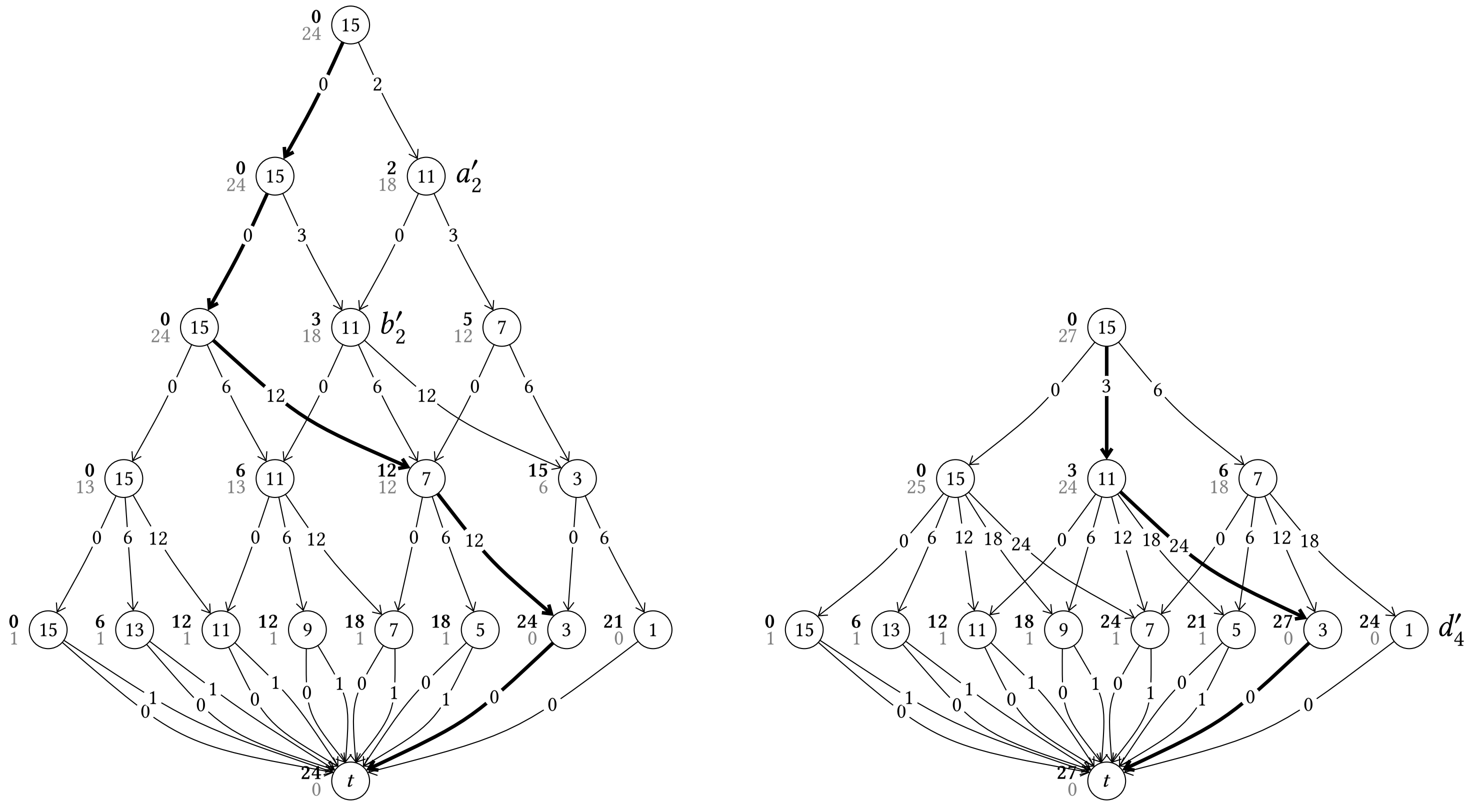
we define two mapping functions: $\Phi : E \longrightarrow \{0, \dots, K-1\}$ that gives the cluster for each original element and $\Phi^{-1} : \{0, \dots, K-1\} \longrightarrow 2^E$ that gives the set of original elements for a given cluster.

Example 5.2.1. *Let us illustrate the problem instance aggregation with our BKP running example. We provide two different complete aggregation schemes with different properties, which are representative for the types of formulations presented in Section 5.4.*

Aggregation (A) *The first aggregation attempts to create a relaxed and simpler BKP instance by reducing the number of distinct item weights. Indeed, states of the DP model for the BKP are identified by the remaining capacity of the knapsack, so having many items with equal weights increases the chances of having different branches overlap. We will thus create K clusters of items based on their similarity and lower the weight of item to the minimum weight among the cluster so that all the solutions are preserved. Formally, we specify* $\Pi(\mathcal{P} = (N, W, V, Q)) = (N, \Pi_W(W), V, Q)$ *with* $\Pi_W(W) = W'$ *with* $W'_i = \min_{k \in \Phi^{-1}(\Phi(i))} W_i$ *for all* $i \in N$.

Let us apply this aggregation to the instance of Table 2.1 with $K = 3$ *and the following clustering:* $\Phi(0) = 0, \Phi(1) = 1, \Phi(2) = 1, \Phi(3) = 2, \Phi(4) = 0$ *or equivalently* $\Phi^{-1}(0) = \{0, 4\}, \Phi^{-1}(1) = \{1, 2\}, \Phi^{-1}(2) = \{3\}$. *Table 5.1 gives the aggregate instance with the adapted weights. For instance, the weight of aggregate item 1 is given by:* $W'_1 = \min\{W_1, W_2\} = \min\{6, 4\} = 4$. *Furthermore, Figure 5.2(a) depicts the exact DD compiled for this aggregate instance. As one can notice, this DD has fewer nodes than the exact DD for the original instance pictured on Figure 2.1. Yet, the best solution found has a value of 24 and gives an upper bound for the original instance that is actually the optimal value of the problem.*

Aggregation (B) *A second strategy consists in reducing the number of items of the original instance. It decreases the number of variables and the number of*



(a) Exact DD for aggregation (A)

(b) Exact DD for aggregation (B)

Figure 5.2: Exact DDs compiled for two aggregate instances based on the BKP instance of Table 2.1.

distinct item weights as well. Again, if we create K clusters of items based on their similarity, we can create aggregate items having the lowest weight and the highest value among the items of in their respective clusters. This is formulated as follows: $\Pi(\mathcal{P} = (N, W, V, Q)) = (\Pi_N(N), \Pi_W(W), \Pi_V(V), \Pi_Q(Q))$ with the aggregate items given by $\Pi_N(N) = \{0, \dots, K-1\}$, their weight $\Pi_W(W) = W'$ with $W'_k = \min_{i \in \Phi^{-1}(k)} W_i$ for all $k \in N'$, value $\Pi_V(V) = V'$ with $V'_k = \max_{i \in \Phi^{-1}(k)} V_i$ for all $k \in N'$ and quantity $\Pi_Q(Q) = Q'$ with $Q'_k = \sum_{i \in \Phi^{-1}(k)} Q_i$ for all $k \in N'$.

Assuming the original items are clustered as follows: $\Phi(0) = 0, \Phi(1) = 0, \Phi(2) = 1, \Phi(3) = 1, \Phi(4) = 2$ or equivalently $\Phi^{-1}(0) = \{0, 1\}, \Phi^{-1}(1) = \{2, 3\}, \Phi^{-1}(2) = \{4\}$. Then, the aggregate instance is given by Table 5.2. The weight of the aggregate item 0 is computed as $W'_0 = \min\{W_0, W_1\} = \min\{4, 6\} = 4$, its value as $V'_0 = \max\{V_0, V_1\} = \min\{2, 3\} = 3$ and its quantity as $Q'_0 = Q_0 + Q_1 = 1 + 1 = 2$. Figure 5.2(b) shows the exact DD for this aggregate instance, providing an upper bound of 27 for the original problem.

5.2.2 State Aggregation and Upper Bound

A second mapping function accompanies the problem instance aggregation operator: the *state aggregation operator* $\pi : \mathcal{S} \longrightarrow \mathcal{S}'$ that projects each state of the state space $\mathcal{S}$ of the original problem in the aggregate state space $\mathcal{S}'$. 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. If the aggregation operators Π and π a correctly specified, a relaxation of the subproblem associated with any node u in $\mathcal{B}$ the exact DD for $\mathcal{P}$ can be obtained by solving its aggregate equivalent $\pi(\sigma(u))$.

Proposition 5.2.1. *Given $\mathcal{B}$ the exact DD for problem $\mathcal{P}$, the aggregation operators Π and π define a valid relaxation for every subproblem of $\mathcal{P}$ if and only if for any exact node $u \in \mathcal{B}$, the compilation of a DD $\mathcal{B}'_{u'}$ for problem $\Pi(\mathcal{P})$ and rooted at u' with $\sigma(u') = \pi(\sigma(u))$ verifies that for any $u \rightsquigarrow t$ path $p \in \mathcal{B}$ there exists a path $p' \in \mathcal{B}'_{u'}$ such that $\Delta(x(p)) = x(p')$ and $v(p) \leq v(p')$.*

Definition 5.2.1 (Aggregation-based upper bound). *Given the exact DD $\mathcal{B}$ for a problem $\mathcal{P}$ and aggregation operators Π and π respecting Proposition 5.2.1, the aggregation-based upper bound (AggB) of a node $u \in \mathcal{B}$ is given by $\overline{v}_{agg}(u) = v^*(\mathcal{B}'_{u'})$ with $\mathcal{B}'_{u'}$ the exact DD compiled for problem $\Pi(\mathcal{P})$ from a node u' such that $\sigma(u') = \pi(\sigma(u))$.*

One way of exploiting this aggregation-based relaxation would be to use it as a replacement for 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. Assuming the aggregation operators

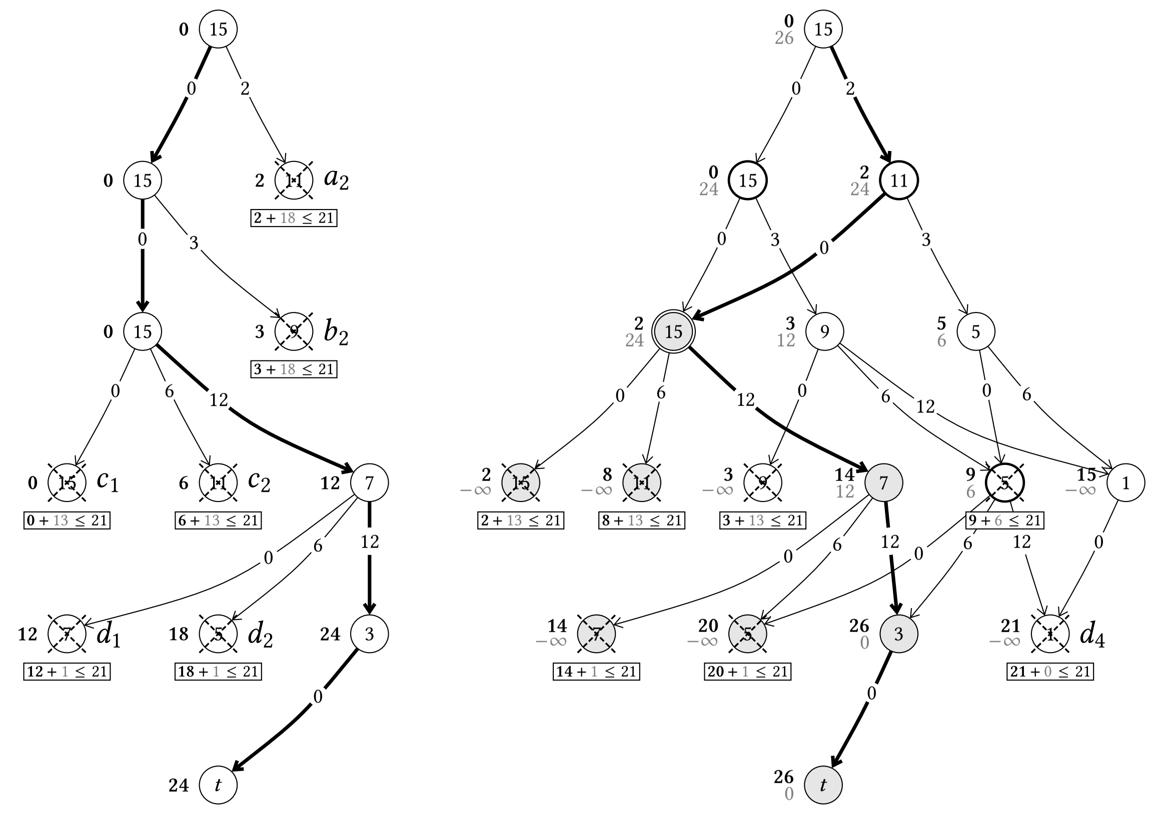
Algorithm 12 Compilation of DD $\mathcal{B}$ rooted at node u_r with max. width W and aggregation-based bounds and heuristic.

```
1: $i \leftarrow$ index of the layer containing $u_r$
2: $L_i \leftarrow \{u_r\}$
3: $\widetilde{D} \leftarrow \Delta^{-1}(\widetilde{d})$ with $\widetilde{d}$ the optimal decisions for $\pi(\sigma(u_r))$   // retrieve AggH
4: for $j = i$ to $n-1$ do
5:    $pruned \leftarrow \emptyset$
6:    perform dominance pruning using Algorithm 7
7:    perform cached-based pruning using Algorithm 10
8:    $L'_j \leftarrow L_j \setminus pruned$
9:    if $|L'_j| > W$ then
10:      restrict or relax the layer to get $W$ nodes with Algorithm 13
11:   $L_{j+1} \leftarrow \emptyset$
12:   for all $u \in L'_i$ do
13:      $\overline{v}_{rub}(\sigma(u)) \leftarrow \min\left\{\overline{v}_{rub}(\sigma(u)), \overline{v}_{agg}(\pi(\sigma(u)))\right\}$   // inject AggB
14:      if $v^*(u \mid \mathcal{B}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}$ then   // RUB pruning
15:         continue
16:      for all $d \in \mathcal{D}_j$ do
17:         create node $u'$ with state $\sigma(u') = t_j(\sigma(u), d)$
            or retrieve it from $L_{j+1}$
18:         create arc $a = (u \xrightarrow{d} u')$ with $v(a) = h_j(\sigma(u), d)$ and $l(a) = d$
19:         $score(a) \leftarrow 1$ if $d \in \widetilde{D}_j$, 0 otherwise
20:         add $u'$ to $L_{j+1}$ and add $a$ to $A$
21: merge nodes in $L_n$ into terminal node $t$
```

are defined such that the aggregate problem is simple enough, we propose to pre-solve it exactly, i.e. compile the associated exact DD, and store the solution of each subproblem – provided by the LocB in the exact DD. By doing so, the AggB can be retrieved very quickly and thus be incorporated in the RUB as shown at line 13 of Algorithm 12 so that it is used as often as possible.

Example 5.2.2. *We continue to illustrate the framework on the BKP problem by providing a state aggregation operator for each aggregation scheme started in Example 5.2.1 and showcasing the effect of the AggB in both cases. To simplify the formulation of the state aggregation operators, we extend the BKP state definition by including the item to consider next. States are thus pairs $\langle c, j\rangle$ with c the remaining capacity and j the next item. The root state is thus $\hat{r} = \langle C, 0\rangle$.*

Aggregation (A) *Since the only effect of the first problem instance aggregation described in Example 5.2.1 is to modify some item weights, the state aggregation*



(a) Relaxed DD with AggB (A) (b) Relaxed DD with AggB (B)

Figure 5.3: Relaxed DDs compiled for the BKP instance of Table 2.1, given the lower bound of 21 obtained in Figure 2.2(a) and the AggB of two different aggregations.

operator can simply be the identity function: $\pi(\langle c, j\rangle) = \langle c, j\rangle$. However, if the AggB is retrieved from a precomputed exact DD $\mathcal{B}'$ for the aggregate instance – as given by Figure 5.2(a) – then there might not exist a state in $\mathcal{B}'$ for each remaining capacity reached in the original problem, because the item weights have been changed. A workaround is to map each original state to an aggregate state with a greater remaining capacity instead. This will also result in a valid upper bound, as formally motivated by the dominance rule presented in Chapter 3. Formally, we write this as

$$\pi(\langle c, j\rangle) = \min\{\langle c', j\rangle \mid \exists u' \in \mathcal{B}' \textit{ such that } \sigma(u') = \langle c', j\rangle \textit{ and } c' \geq c\}.$$

Figure 5.3(a) shows the relaxed DD compiled for the BKP instance of Table 2.1, using the AggB on top of the RUB and provided the lower bound of 21 obtained in Figure 2.2(a). For each node, the AggB is retrieved by finding the node with the corresponding aggregate state in Figure 5.2(a). The AggB allows pruning nodes a_2 and b_2 and this additional filtering produces an exact DD, therefore finding and proving the optimal solution at the root node. The AggB of nodes a_2 and b_2 is respectively given by nodes a_2' and b_2' of Figure 5.2(a). For node a_2, we have that $\sigma(a_2) = \sigma(a_2')$. However, Figure 5.2(a) does not contain any node with state $\sigma(b_2) = \langle 9, 2\rangle$ at the same layer. Node b_2' with $\sigma(b_2') = \langle 11, 2\rangle$ is thus used instead and still provides an AggB that allows to prune b_2. For nodes c_1, c_2, d_1 and d_2, the AggB is equal to the RUB and does not generate additional pruning.

Aggregation (B) *The state aggregation operator are similar to those of aggregation (A), except that we need to adapt the item that is considered next since the aggregate instances deals with fewer aggregate items. The aggregate item to consider next is given by the cluster of the original item to consider next, and thus:*

$$\pi(\langle c, j\rangle) = \min\{\langle c', \Phi(j)\rangle \mid \exists u' \in \mathcal{B}' \textit{ such that } \sigma(u') = \langle c', \Phi(j)\rangle \textit{ and } c' \geq c\}.$$

An important detail of this aggregation scheme is that only set of consecutive original items can be clustered together so that the set of remaining items to consider in the original problem is coherent with the set of aggregate items still to consider.

Given the AggBs computed from Figure 5.2(b), we can compile the relaxed DD of Figure 5.3(b). This time, only node d_4 is pruned by the AggB, which is given by node d_4' at layer $\Phi(4) = 2$ of Figure 5.2(b).

5.2.3 Solution Disaggregation and Node Selection Heuristic

The AggB is not the only information that we can extract from the solution of an aggregate subproblem. Indeed, besides the value of this solution, the

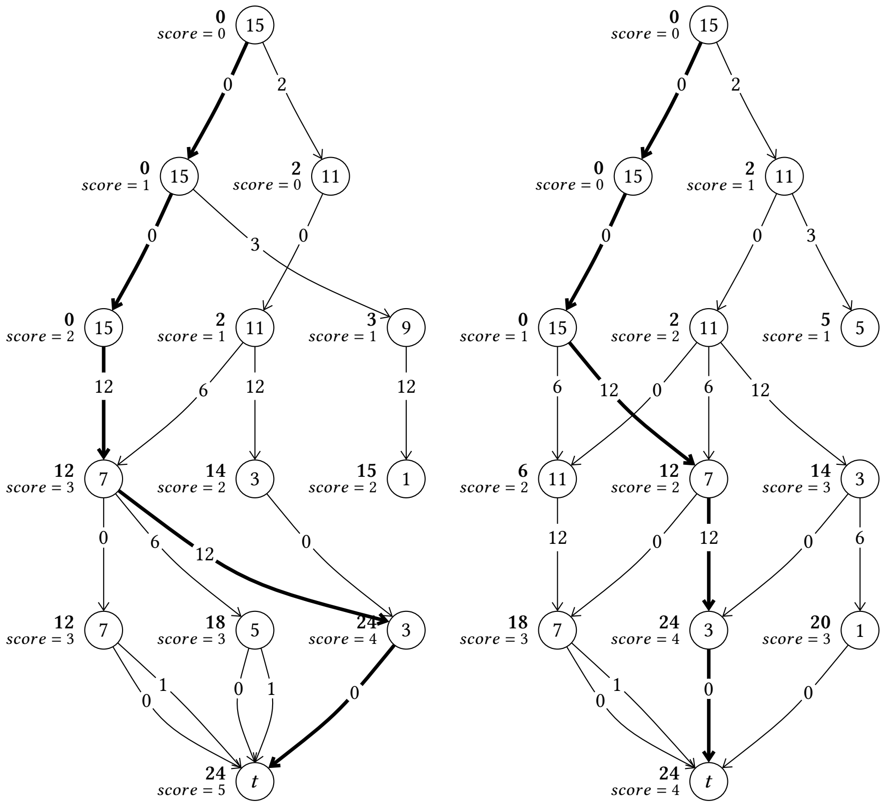
Algorithm 13 Restriction or relaxation of layer L_j with maximum width W.

```
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 $\sigma(\mu) = \oplus(\sigma(\mathcal{M}))$        // for relaxation only
       and add it to $L_j$
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 $v(a') = \Gamma_{\mathcal{M}}(v(a), u)$
```

aggregate decisions that constitute it can also be recorded. If the correspondence between decisions made for the aggregate problem with decisions for the original problem is known, then the solution of the aggregate version of a subproblem can be used to find good heuristic solutions for the original subproblem. Concretely, we propose an *aggregation-based node selection heuristic* (AggH) that steers the compilation of restricted DDs towards solutions resembling the optimal aggregate one. A last modeling component is needed for this: the *decision disaggregation operator* $\delta^{-1}(d) : \mathcal{D}'_k \to 2^{\mathcal{D}_i} \times \cdots \times 2^{\mathcal{D}_j}$ 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: $\Delta^{-1}(p = (d_k, \ldots, d_{n'-1})) = \delta^{-1}(d_k) \cdot \ldots \cdot \delta^{-1}(d_{n'-1})$ where n' is the supposed number of aggregate variables. Using this operator, we can compute a *score* for each decision made during the compilation of restricted DDs. At line 3 of Algorithm 12, 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 19, depending on its compatibility with the disaggregated solution. At line 2 of Algorithm 13, 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.

Example 5.2.3. *As for the AggB, we provide a decision disaggregation operator for each BKP aggregation scheme developed throughout the chapter and illustrate its use.*

Aggregation (A) *The decision disaggregation operator yields a single decision with a single possibility $\delta^{-1}(d) = (\{d\})$ since there is a one-to-one mapping*



(a) Restricted DD with AggH (A) (b) Restricted DD with AggH (B)

Figure 5.4: Restricted DDs compiled for the BKP instance of Table 2.1, given the AggH of two different aggregations.

between original and aggregate items. Therefore, the optimal solution of the aggregate instance $x' = (0, 0, 2, 2, 0)$ *obtained in Figure 5.2(a) is disaggregated as* $\Delta^{-1}(x') = (\{0\}, \{0\}, \{2\}, \{2\}, \{0\})$. *Since it corresponds to the optimal solution of the original problem, using it as a node selection heuristic can guide the compilation of a restricted DD towards this optimal solution, as shown by Figure 5.4(a).*

Aggregation (B) *The decision disaggregation operator for the second aggregation scheme is less simple since a single aggregate decision corresponds to multiple original items. A possible way to disaggregate a decision* d_k *concerning the aggregate item* k *corresponding to the original items* $i, \ldots, j$ *is to select as many copies of item* i, *then* $i + 1$, *and so on until having selected* d_k *copies in total. Formally, this formulated as follows:* $\delta^{-1}(d_k) = \left(\{d_i\}, \ldots, \{d_j\}\right)$

Algorithm 14 The restricted DD-based branch-and-bound algorithm.

```
1: $Fringe \leftarrow \{r\}$        // a priority queue ordered by decreasing $v(u) + \overline{v}(u)$
2: $Fronts_j \leftarrow \emptyset$ for $j = 0, \dots, n$ // hash tables of dom. keys to Pareto fronts
3: $Cache \leftarrow \emptyset$  // a hash table of states to threshold $\sigma(u) \rightarrow \langle \theta, expanded \rangle$
4: $\underline{x} \leftarrow \bot, \underline{v} \leftarrow -\infty$                    // incumbent solution and its value
5: while $Fringe$ is not empty do
6:    $u \leftarrow$ best node from $Fringe$, remove it from $Fringe$
7:    if $v(u) + \overline{v}(u) \leq \underline{v}$ then
8:       continue
9:    if $Cache.contains(\sigma(u))$ then
10:      $\langle \theta, expanded \rangle \leftarrow Cache.get(\sigma(u))$
11:      if $v(u) < \theta$ or $(v(u) = \theta \wedge expanded)$ then
12:         continue
13:   $\underline{\mathcal{B}} \leftarrow Restricted(u)$        // compile restricted DD with Algorithm 12
14:   if $v^*(\underline{\mathcal{B}}) > \underline{v}$ then                        // update incumbent
15:      $\underline{x} \leftarrow x^*(\underline{\mathcal{B}}), \underline{v} \leftarrow v^*(\underline{\mathcal{B}})$
16:   if $\underline{\mathcal{B}}$ is not exact then
17:      update $Cache$ with Algorithm 15 applied to $\underline{\mathcal{B}}$
18:      for all $u' \in LEL(\underline{\mathcal{B}})$ do
19:         $v(u') \leftarrow v^*(u' \mid \underline{\mathcal{B}}), \overline{v}(u') \leftarrow \overline{v}_{rub}(\sigma(u')), p(u') \leftarrow p^*(u' \mid \underline{\mathcal{B}})$
20:         if $v(u') + \overline{v}(u') > \underline{v}$ then
21:            add $u'$ to $Fringe$
22: return $(\underline{x}, \underline{v})$
```

with $\{i, \dots, j\} = \Phi^{-1}(k)$ *and* $d_l = \max\left\{0, \min\left\{Q_l, d_k - \sum_{\substack{m \in \Phi^{-1}(k) \\ m<l}} Q_m\right\}\right\}$ *for* $l = i, \dots, j$.

The optimal solution of the aggregate instance $x' = (1, 4, 0)$ *obtained in Figure 5.2(b) is thus disaggregated as* $\Delta^{-1}(x') = (\{1\}, \{0\}, \{2\}, \{2\}, \{0\})$. *Even if it does not perfectly correspond to the optimal solution of the problem, it resembles it sufficiently for the corresponding AggH to prevent it from being removed when compiling a restricted DD, as shown by Figure 5.4(b).*

5.3 Restricted Branch-and-Bound

So far, we have suggested that the AggB could be used to complement the bounds provided by relaxed DDs. Another possibility is to completely replace the state merging-based relaxation and only rely on AggBs and RUBs to provide dual bounds for cutset nodes. In the case where the AggBs are precomputed, the B&B algorithm can be revised to only manipulate restricted DDs. We refer to this variant as the *restricted B&B* and formally define it with

Algorithm 15 Computation of the threshold $\theta(u \mid \underline{\mathcal{B}})$ of every node u above the LEL of a restricted DD $\underline{\mathcal{B}}$ and update of the *Cache*.

```
1: $i \leftarrow$ index of the root layer of $\underline{\mathcal{B}}$, $lel \leftarrow$ index of the LEL of $\underline{\mathcal{B}}$
2: $(L_i, \dots, L_n) \leftarrow Layers(\underline{\mathcal{B}})$
3: $\theta(u \mid \underline{\mathcal{B}}) \leftarrow \infty$ for all $u \in \underline{\mathcal{B}}$
4: for $j = lel$ down to $i$ do
5:   for all $u \in L_j$ do
6:     if $Cache.contains(\sigma(u))$ and $v^*(u \mid \underline{\mathcal{B}}) \leq \theta(Cache.get(\sigma(u)))$ then
7:       $\theta(u \mid \underline{\mathcal{B}}) \leftarrow \theta(Cache.get(\sigma(u)))$
8:     else
9:       if $\exists \Psi' \in Fronts_j[\kappa(u)]$ s.t. $\Psi' \geq \Psi(u)$ then        // dom. pruning
10:        $\theta(u \mid \underline{\mathcal{B}}) \leftarrow v' - 1$ if $\psi' = \psi(u)$ else $v'$ (with $\Psi' = (v') \cdot \psi'$)
11:      else if $v^*(u \mid \underline{\mathcal{B}}) + \overline{v}_{rub}(\sigma(u)) \leq \underline{v}$ then        // RUB pruning
12:        $\theta(u \mid \underline{\mathcal{B}}) \leftarrow \underline{v} - \overline{v}_{rub}(\sigma(u))$
13:      else if $u \in LEL(\underline{\mathcal{B}})$ then
14:        $\theta(u \mid \underline{\mathcal{B}}) \leftarrow v^*(u \mid \underline{\mathcal{B}})$
15:      $expanded \leftarrow$ false if $u \in LEL(\underline{\mathcal{B}})$ else true
16:      $Cache.insertOrReplace(\sigma(u), \langle\theta(u \mid \underline{\mathcal{B}}), expanded\rangle)$
17:    for all arc $a = (u' \xrightarrow{d} u)$ incident to $u$ do
18:      $\theta(u' \mid \underline{\mathcal{B}}) \leftarrow \min\{\theta(u' \mid \underline{\mathcal{B}}), \theta(u \mid \underline{\mathcal{B}}) - v(a)\}$
```

Algorithm 14. It proceeds exactly like the classical B&B algorithm, except that no relaxed DDs are compiled and that ECs are thus extracted from restricted DDs. We only define the LEL type of EC for restricted DDs, which is given by the deepest layer that was not restricted during the top-down compilation.

The second difference concerns the computation of the expansion thresholds, as described in Algorithm 15. This modified algorithm simply assigns a dominance or pruning threshold to each node of the LEL and performs the bottom-up propagation for all nodes above the LEL, as explained in Chapter 4. Note that the pruning thresholds are only derived from dominance relations and RUB pruning, as no LocB pruning occurs in restricted DDs.

5.4 Applications

In this section, we provide the aggregation operators for three optimization problems that will serve to illustrate and evaluate the impact of our proposed framework.

5.4.1 Talent Scheduling Problem

In [GSC11], 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, this leads to the following aggregation operators.

- Problem instance aggregation: it is defined as $\Pi(\mathcal{P} = (N, A, R, D, C)) = (\Pi_N(N), A, \Pi_R(R), \Pi_D(D), C)$ with $\Pi_N(N) = \{0, \dots, K-1\}$. 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 $i \in \Pi_N(N)$.

- State aggregation operator: this operator is less straightforward to define because aggregate states are only defined in terms of complete aggregated scenes that have yet to be scheduled. Therefore, we restrict the state aggregation operator to states $s \in \mathcal{S}$ that must still schedule a subset of scenes that exactly corresponds to a subset of aggregate scenes, i.e. $s.P = \emptyset$ and for all $k = 0, \dots, K-1$, either $\Phi^{-1}(k) \subseteq s.M$ or $\Phi^{-1}(k) \cap s.M = \emptyset$. For such states s, the state aggregation operator simply computes the corresponding set of aggregate scenes: $\pi(s) = (M', \emptyset)$ with $M' = \left\{ i \in \Pi_N(N) \mid \Phi^{-1}(i) \subseteq s.M \right\}$.

- Disaggregation: each aggregate scene corresponds to a set of original scenes, we thus need to map each aggregate decision to a sequence of original decisions: $\delta^{-1}(i) = V$ 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.

5.4.2 Pigment Sequencing Problem

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 optimally by a greedy algorithm. 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.

- Problem instance aggregation: the operator is formulated as $\Pi(\mathcal{P} = (I, S, C, H, Q)) = (\Pi_I(I), \Pi_S(S), \Pi_C(C), H, \Pi_Q(Q))$, where the aggregate set of item types is given by $\Pi_I(I) = \{0, \dots, K-1\}$. 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 pairwise 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.

- State aggregation: 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.

- Disaggregation: this operator is straightforward 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: $\delta^{-1}(i) = \left(\Phi^{-1}(i)\right)$.

5.4.3 Aircraft Landing Problem

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 aggregation operators as follows.

- Problem instance aggregation: it involves defining several components $\Pi(\mathcal{P} = (N, R, C, A, S, T, L)) = (N, R, \Pi_C(C), \Pi_A(A), \Pi_S(S), T, \Pi_L(L))$. The set of aircraft, their target landing time and the number of runways is unchanged. The aggregate set of classes is given by $\Pi_C(C) = \{0, \dots, K-1\}$ and their corresponding set of aircraft 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 separation time between aggregate classes is computed as the minimum separation time between any two classes belonging to the respective clusters, 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 aircraft so that any aircraft with a given target landing time has a greater latest landing time than all other aircraft of the same class with a smaller target landing time: $\Pi_L(L) = L'$ with $L'_i = \max\left\{L_j \mid \Phi(i) = \Phi(j), T_i \leq T_j\right\}$ for all $i \in A$. This property is assumed to hold for the original problem instance, and must be preserved so that aircraft from the same class can be scheduled sequentially in the DP model.

- State aggregation: in the same fashion as for the PSP, assuming $\Phi(\perp) = \perp$, the state aggregation operator is defined by $\pi(s) = (Q', ROP')$ with the remaining quantities of aircraft 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 $i \in R$.

  In Example 5.2.2, we explained that after pre-solving an aggregate BKP instance with modified items, states reached during the resolution of the original instance might always not match the precomputed ones. The same phenomenon arises for the ALP because separation times are changed and therefore lead to states with very different previous landing times on each runway. However, as for the BKP, 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 $i \in R$.

- Disaggregation: 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: $\delta^{-1}(a, r) = \left\langle \left\{ (a', r) \mid a' \in \Phi^{-1}(a) \right\} \right\rangle$.

5.5 Computational Experiments

The impact of the aggregation-based bounds and heuristics was evaluated experimentally by extending DDO [GSC21] and injecting the modeling of the three discrete optimization problems presented in Section 5.4. As described in Sections 3.4 and 4.5, 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 4 aggregate item types. On the other hand, not all ALP instances reduced to 2 aggregate aircraft classes have a reasonable number of states so we employ a relaxed DD with maximum width 40000 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 [GSC11]. Six different configurations were created by combining DDO and rDDO – using the restricted B&B – with the aggregation-based bounds (AggB) and heuristics (AggH). All configurations used the caching mechanism presented in Chapter 4. Ten minutes were allotted for each configuration to solve each instance.

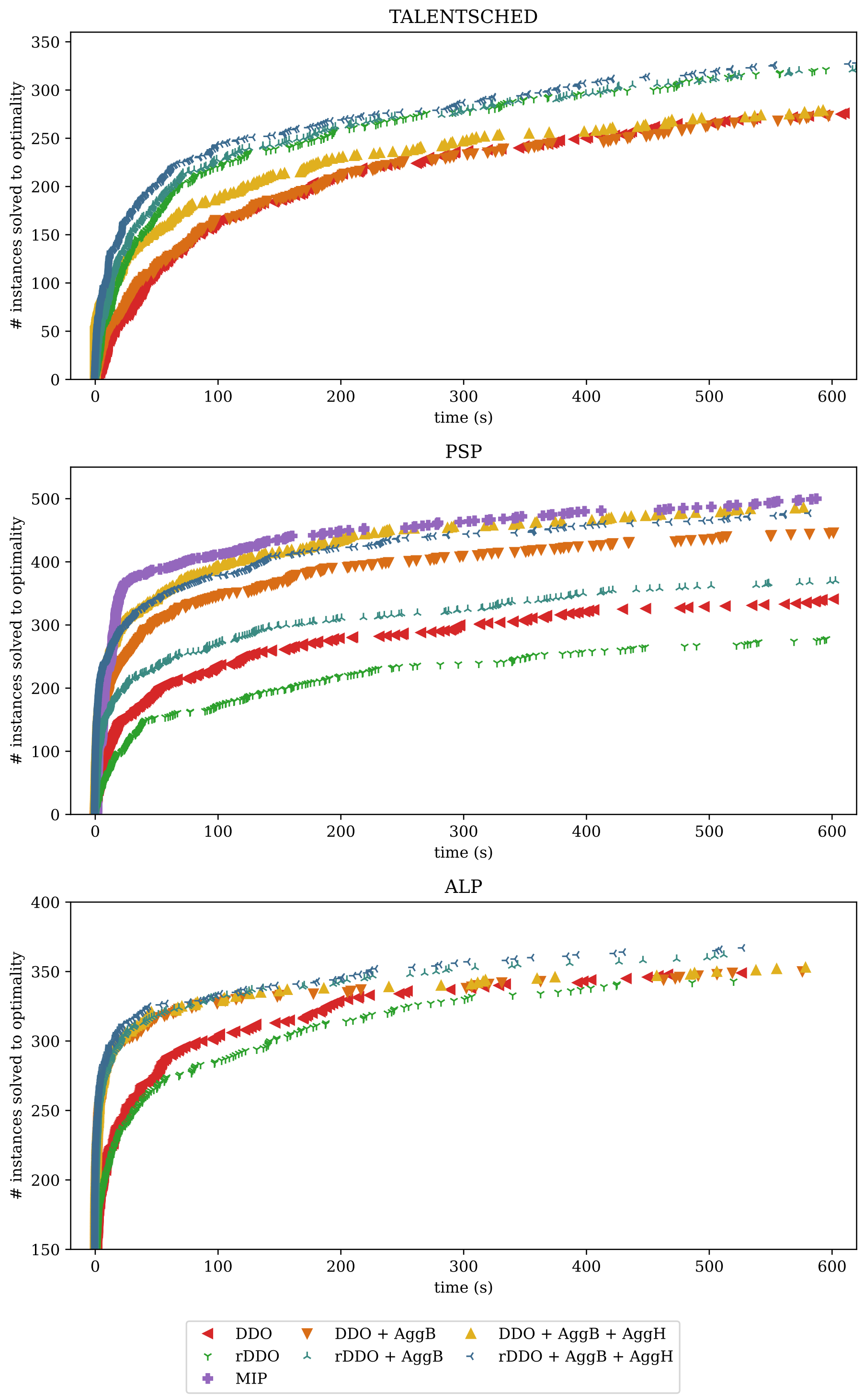


Figure 5.5: Number of instances solved over time for each configuration and problem studied.

5.5.1 Number of Benchmark Instances Solved

Figure 5.5 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 not trivial – see [GSC11] – and generates a lot of pruning. Still, adding the AggB and the AggH to either configuration improves the results by a small margin, especially in the case of the AggH.

For the PSP, the impact of the AggB and AggH is much more significant: adding the AggB to DDO allows solving 105 additional instances, and when also activating the AggH, 40 extra instances are solved, which is only 14 fewer than Gurobi out of the 648 benchmark instances. Moreover, while rDDO alone yields the worst results, incorporating the AggB leads to results that are already better than those achieved by DDO. Combining it with the AggH performs even better, and actually almost equally well than DDO+AggB+AggH, solving 477 instances compared to 486 for DDO+AggB+AggH.

Finally, we can observe that the number of ALP instances solved overall by all the configurations is not dramatically different. Using the AggB with DDO helps to solve only one additional instance, and the AggH four more. This difference is more significant for rDDO, where 19 additional instances are closed when integrating the AggB, and 5 more with the AggH. Furthermore, rDDO+AggB+AggH solves 13 instances more than DDO+AggB+AggH, which again shows that relaxed DDs might not provide bounds whose quality justifies the computational effort required to obtain them for this particular DP model. For both variants of the solver, we can also notice that, even if the total number of instances solved is not very far apart, the AggB allows many of those instances to be solved more quickly.

5.5.2 End Gap

The impact of the AggB and the AggH can also be measured in terms of the end gap obtained for all instances, including unsolved ones. The end gap is computed as: $\frac{UB-LB}{UB}$. The left column of Figure 5.6 compares the end gap obtained for each instance by DDO and DDO+AggB+AggH. For the large majority of the instances, the end gap is smaller when using the AggB and the AggH, which means that DDO+AggB+AggH is closer to terminating the search much more often than DDO, especially for the PSP.

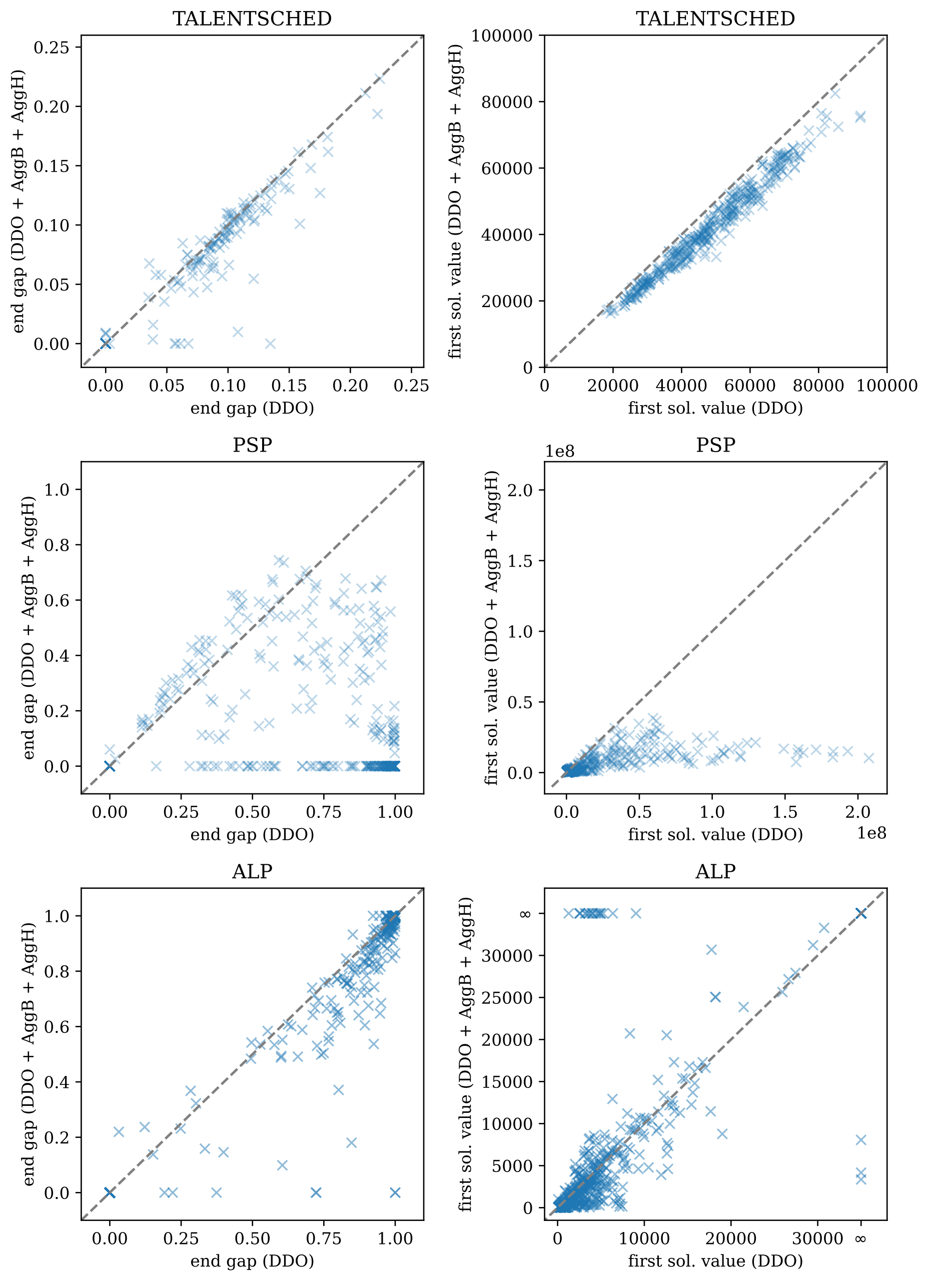


Figure 5.6: Comparison of the end gap and the value of the first solution obtained for each instance by DDO and DDO+AggB+AggH.

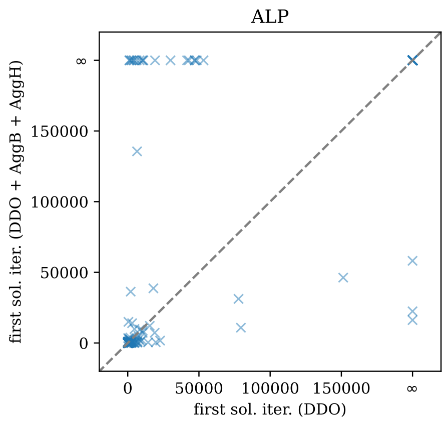


Figure 5.7: Comparison of the iteration at which the first solution is found by DDO and DDO+AggB+AggH for ALP.

5.5.3 Quality of the First Solution

To validate the relevance of the AggH, we also compare the value of the first solution found by DDO and DDO+AggB+AggH in the right column of Figure 5.6. For all problems, the quality of the first solution is almost always better when using the AggH, and in many cases by a large margin for PSP instances. It is actually better for all 400 TalentSched instances. For the PSP, the first solution has a better value in 481 cases when using the AggB, and in 166 cases without it. However, we can see on Figure 5.6 that in those cases, the first solution found with the AggH is never far from the first solution found without it. For the ALP, a better solution is obtained using the AggH for 300 instances, and a worse one for 142 instances. Unlike for the TalentSched and the PSP, for which a solution is always found at the first iteration, the landing time windows of the ALP make it difficult to find a feasible solution. We thus also compare on Figure 5.7 the iteration at which the first solution is found. This may be difficult to observe on the graph, but DDO+AggB+AggH finds a feasible solution earlier than DDO in 174 cases, compared to 64 cases in the opposite direction. Yet, for 19 instances where DDO manages to find a feasible solution, DDO+AggB+AggH fails to find a single one of them, whereas there are only 3 instances for which DDO+AggB+AggH could find a solution and DDO could not. This shows that the AggH can also provide misleading information that causes restricted DDs to miss all good solutions when the aggregation does not represent the original instance well enough. Still, overall the AggH proves to be useful much more often than not, and it would suffice to avoid relying on it for certain restricted DDs to avoid being penalized when it does not lead to good solutions.

5.6 Conclusion

This chapter 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 can 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. However, there were also a few cases in the experiments where the aggregate solution did not correspond to original feasible solutions very well, and thus the aggregation-based node selection heuristic caused the solver to repeatedly miss good solutions. Nevertheless, these results suggest that aggregation-based bounds and heuristics can capture global problem structures well, sometimes much better than the state merging-based relaxation and the greedy *MinLP* heuristic classically used to compile approximate DDs.

The Constrained Single-Row Facility Layout Problem

6

This chapter is largely based on the following paper: V. Coppé, X. Gillard, and P. Schaus. "Solving the Constrained Single-Row Facility Layout Problem with Decision Diagrams". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022. It presents how additional constraints can be added to existing MIP and DP models for the *Single-Row Facility Layout Problem*. It also presents an RLB for this problem.

6.1 Introduction

The *Single-Row Facility Layout Problem* (SRFLP) is an ordering problem considering a set of departments in a facility, with given lengths and pairwise traffic intensities. Its goal is to find a linear ordering of the departments minimizing the weighted sum of the distances between department pairs. The SRFLP is applied in different fields to arrange items such as rooms on a corridor in hospitals or offices [Sim69], airplanes and gates in an airport [SGW91], machines in a manufacture [HK88], books on a shelf and files in disk cylinders [PQ81]. When all facilities have equal lengths and the traffic intensities are binary, the problem is known as the *Minimum Linear Arrangement Problem* (MinLA). It is a well-known graph layout problem which has been proved to be NP-hard [GJ79] and consequently, so is the SRFLP.

Due to its difficulty in being solved by exact methods, many heuristic techniques have been designed to find good quality solutions to the SRFLP problem [Dre87; Hal70; HA92; KHL95] and more recently [DAF11; GL16; KG14; Pal17; SE10]. The first attempt to solve the SRFLP optimally was a B&B algorithm with interesting lower bounds [Sim69]. Later, the DP approach presented in [KH67] was applied to the SRFLP in [PQ81]. More recent techniques include non-linear programming [HK91], MIP [Ama06; Ama08;

LW76], branch-and-cut [Ama09; AL13] and semidefinite programming [AKV05; AV08; AY09; HR13a; HR13b].

In [KD18], *positioning*, *ordering* and *relation* constraints were suggested for the SRFLP to model real-life situations. The resulting problem is called the *Constrained Single-Row Facility Layout Problem* (cSRFLP). They also proposed a permutation-based genetic algorithm to solve this new problem and reported very good results, with objective values deviating by only a few percents from the best known solutions to the unconstrained problem for instances with up to 100 departments. In [Liu+21], a MIP model to solve the cSRFLP is introduced and a constrained improved fireworks algorithm is described. The latter is shown to find solutions of better quality than the genetic algorithm of [KD18].

This chapter begins with a formal definition of the SRFLP in Section 6.2 and of the constraints that constitute the cSRFLP. We then present two novel exact models to solve the problem. In Section 6.3, we model the constraints of the cSRFLP on top of the state-of-the-art MIP model for the SRFLP [Ama09]. The almost exact same formulation was presented the same year in [MT23]. Likewise, Section 6.4 recalls the DP model for the SRFLP from [PQ81], shows how the new constraints can be integrated and describes an RLB for this problem. In Section 6.5, computational experiments comparing the two MIP models and the DP model solved with DDO are presented. They show that our two new models outperform the MIP model from [Liu+21] in terms of solving time. Other than that, the DD-based approach and the new MIP model produce similar results. The former seems to handle positioning constraints better while the latter is particularly efficient for relation constraints. The chapter concludes with a summary of our contributions and directions for future work.

6.2 Problem Definition

This section is organized as follows, a formal definition of the SRFLP is given in Section 6.2.1 which is then completed in Section 6.2.2 with the constraints that constitute the cSRFLP.

6.2.1 SRFLP

The SRFLP is an ordering problem that aims to place a set of departments $N = \{0, \ldots, n-1\}$ on a line. Each department $i \in N$ has a positive length L_i, and the connection between each pair of departments $i, j \in N$ is described by a positive traffic intensity C_{ij}. It is imposed that $C_{ij} = C_{ji}$ but it is not a modeling restriction since a trip in any direction covers the same distance, the traffic intensities can thus concentrate both directions [Sim69].

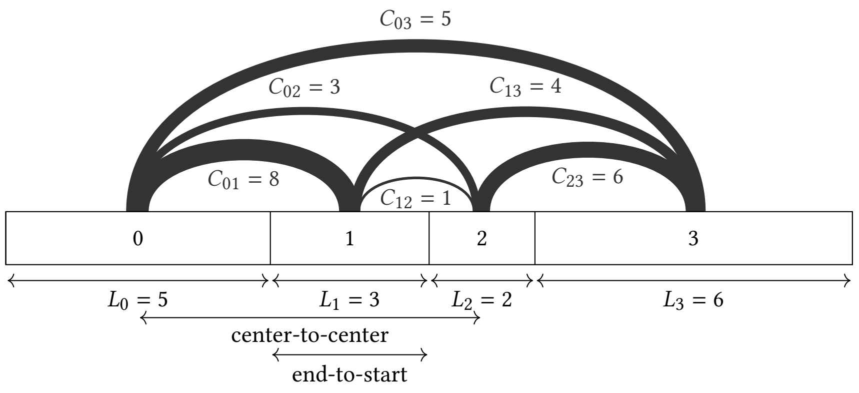


Figure 6.1: An instance of the SRFLP with 4 departments ordered optimally. The lengths of the departments are noted below them and the pairwise traffic intensities are given on the edges connecting pairs of departments. Center-to-center and end-to-start distances between departments 0 and 2 are shown.

The goal is to find a bijection $\pi : N \longrightarrow N$ that maps each department to a position on the line, while minimizing the total distance covered in the facility, which is formulated as follows:

$$SRFLP(\pi) = \sum_{k=0}^{n-1} L_k \sum_{\substack{i=0 \\ \pi(i)<\pi(k)}}^{n-1} \sum_{\substack{j=0 \\ \pi(k)<\pi(j)}}^{n-1} C_{ij} + \underbrace{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} C_{ij} \frac{L_i + L_j}{2}}_{K} . \tag{6.1}$$

The second term of Equation (6.1) is a constant that does not depend on the ordering π, which is usually denoted K [Sim69]. It accounts for the contributions of the half department lengths in each pairwise center-to-center distance.

Example 6.2.1. *Let us illustrate the computation of the objective function on the facility given by Figure 6.1. We first compute the value of the constant K:*

$$\begin{aligned} K &= C_{01}\frac{L_0 + L_1}{2} + C_{02}\frac{L_0 + L_2}{2} + C_{03}\frac{L_0 + L_3}{2} + C_{12}\frac{L_1 + L_2}{2} + C_{13}\frac{L_1 + L_3}{2} + C_{23}\frac{L_2 + L_3}{2} \\ &= 8\frac{5+3}{2} + 3\frac{5+2}{2} + 5\frac{5+6}{2} + 1\frac{3+2}{2} + 4\frac{3+6}{2} + 6\frac{2+6}{2} \\ &= 8 \times 4 + 3 \times 3.5 + 5 \times 5.5 + 1 \times 2.5 + 4 \times 4.5 + 6 \times 4 = 114.5 \end{aligned}$$

and then the cost of the ordering $\pi(i) = i, \forall i \in N$ as shown in Figure 6.1:

$$\begin{aligned} SRFLP(\pi) &= C_{01} \times 0 + C_{02}L_1 + C_{03}(L_1 + L_2) + C_{12} \times 0 + C_{13}L_2 + C_{23} \times 0 + K \\ &= 8 \times 0 + 3 \times 3 + 5 \times (3 + 2) + 1 \times 0 + 4 \times 2 + 6 \times 0 + 114.5 = 156.5. \end{aligned}$$

6.2.2 cSRFLP

The cSRFLP is obtained by adding three types of constraints to the SRFLP:

- *Positioning constraints:* A department is forced to be located at a specific position within the ordering. These constraints are described by a function $position : N \longrightarrow N \cup \{\perp\}$ which maps positions to their corresponding department or to $\perp$ if there is no constraint on the position. To simplify the coming equations, we also define the function $department : N \longrightarrow N \cup \{\perp\}$ which is the inverse mapping, between departments and positions.
- *Ordering constraints:* These constraints impose that some department must come before another one in the ordering. Formally, the function $predecessors : N \longrightarrow 2^N$ gives the set of *predecessors* of each department, i.e. all departments that must be placed on the left of the given department.
- *Relation constraints:* Similarly to ordering constraints, relation constraints impose a relative ordering between a pair of departments. In this case, however, the two departments are required to be adjacent in the ordering. The function $previous : N \longrightarrow N \cup \{\perp\}$ maps departments to the department that must be placed right before, or to $\perp$ if there is no such constraint.

6.3 Mixed-Integer Programming Model

In this section, we integrate the constraints of the cSRFLP to the MIP model for the SRFLP, used within the branch-and-cut framework of [Ama09]. This model uses betweenness variables ζ_{ijk} which describe the relative ordering of departments $i, j, k \in N$ in an ordering π:

$$\zeta_{ijk}^{\pi} = \begin{cases} 1, & \text{if } \pi(i) < \pi(k) < \pi(j) \text{ or } \pi(j) < \pi(k) < \pi(i) \\ 0, & \text{otherwise.} \end{cases} \tag{6.2}$$

Using those variables, the objective function can be formulated as follows:

$$SRFLP(\zeta) = \sum_{i \in N} \sum_{\substack{j \in N \\ i<j}} C_{ij} \sum_{k \in N} \zeta_{ijk} L_k + K \tag{6.3}$$

and is to be minimized under the following constraints:

$$\zeta_{ijk} = \zeta_{jik} \qquad \forall \{i, j, k \mid i < j\} \subseteq N \tag{6.4}$$

$$\zeta_{ijk} + \zeta_{ikj} + \zeta_{jki} = 1 \qquad \forall \{i, j, k\} \subseteq N \tag{6.5}$$

$$\zeta_{ijd} + \zeta_{jkd} - \zeta_{ikd} \geq 0 \qquad \forall \{i, j, k, d\} \subseteq N \tag{6.6}$$

$$\zeta_{ijd} + \zeta_{jkd} + \zeta_{ikd} \leq 2 \qquad \forall \{i, j, k, d\} \subseteq N. \tag{6.7}$$

Equation (6.4) follows from the definition of the betweenness variables in Equation (6.2). Equation (6.5) states that only one department among i, j, k lies between the two others. Finally, Equations (6.6) and (6.7) express the fact that when a department d is placed between departments i and k, then the department d must either lie between departments (a) i and j or (b) j and k, but not both (a) and (b).

We now present how the constraints of the cSRFLP can be integrated in the model. A solution to the original model specifies a relative ordering of the departments. Yet, it does not impose one extremity to the left of the arrangement. As we will need this information in the constraints presented in Section 6.2.2, we solve this issue by adding two dummy departments L and R. For the cSRFLP, the set of departments is thus defined as $N = \{0, \ldots, n-1\} \cup \{L, R\}$ and departments L and R also obey Equations (6.2) to (6.7). We set $L_L = L_R = 0$ and $C_{Li} = C_{iL} = C_{Ri} = C_{iR} = 0, \forall i \in N$ so that the dummy departments have no impact on the objective function. Department L and R are respectively forced on the left and right side of the arrangement by adding the constraints:

$$\zeta_{LRi} = 1 \qquad \forall i \in N \setminus \{L, R\} \tag{6.8}$$

$$\zeta_{ijL} = 0 \qquad \forall i, j \in N \tag{6.9}$$

$$\zeta_{ijR} = 0 \qquad \forall i, j \in N. \tag{6.10}$$

Equation (6.8) imposes that all other departments are placed between departments L and R. Inversely, Equations (6.9) and (6.10) ensure that departments L and R are not placed between any two departments.

We can now write the additional constraints of the model for the cSRFLP:

$$\sum_{k=0}^{n-1} \zeta_{Lik} = j \qquad \forall i \in N, position(i) = j \neq \perp \tag{6.11}$$

$$\sum_{k=0}^{n-1} \zeta_{iRk} = n - j - 1 \qquad \forall i \in N, position(i) = j \neq \perp \tag{6.12}$$

$$\zeta_{Lij} = 0 \quad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{6.13}$$

$$\zeta_{Lji} = 1 \quad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{6.14}$$

$$\zeta_{iRj} = 1 \quad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{6.15}$$

$$\zeta_{jRi} = 0 \quad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{6.16}$$

$$\zeta_{ijk} = 0 \qquad \forall i, j \in N, i = previous(j), k \in N \setminus \{i, j\}. \tag{6.17}$$

Equations (6.11) and (6.12) ensure that j departments are located on the left of department i and $n - j - 1$ on the right, given that i must be placed at the j-th position. Equations (6.13) to (6.16) impose that i is placed between

L and j and that j is placed between i and R, when either i is a predecessor of j or i must be placed right before j. Finally, Equation (6.17) is added for relation constraints to avoid having any departments placed between the two departments involved in the constraint.

6.4 Dynamic Programming Formulation

Section 6.4.1 recalls an efficient DP model for the SRFLP introduced in [PQ81]. We then show in Section 6.4.2 how the constraints can be incorporated in this model. As a whole, this formulation will be the starting point for our DD-based approach.

6.4.1 SRFLP

The idea of the DP model is to place the departments one by one on the line from left to right. From Equation (6.1), it is clear that the individual cost of placing department k at position $\pi(k)$ only depends on the side on which all other departments are located with respect to k. If the state of the DP model is the subset of departments which remain to be placed – called *free* departments from now on, as opposed to *fixed* departments – we can compute this individual cost and recursively find the optimal ordering of each subset of N. Formally, the components of the DP model can be defined as follows.

- Control variables $x_j \in \mathcal{D}_j$ with $j \in \{0, \dots, n-1\}$ decide which department is placed at position j on the line. All variables have the same domain $\mathcal{D}_j = N$ since departments can appear anywhere in the ordering.

- The state space $\mathcal{S}$ contains all subsets of N. In particular, the root state $\hat{r} = N$ contains all departments since no departments have been placed, and inversely, the terminal state $\hat{t} = \emptyset$ is reached when all departments are placed.

- The transition functions simply remove the selected department from the state if present, and point to the infeasible state otherwise:

$$t_j\left(s^j, x_j\right) = \begin{cases} s^j \setminus \{x_j\}, & \text{if } x_j \in s^j \\ \hat{0}, & \text{otherwise.} \end{cases}$$

- The transition value functions are given by:

$$h_j\left(s^j, x_j\right) = \begin{cases} L_{x_j} \sum_{i \in \bar{s}^j} \sum_{k \in s^j \setminus \{x_j\}} C_{ik}, & \text{if } x_j \in s^j \\ 0, & \text{otherwise.} \end{cases}$$

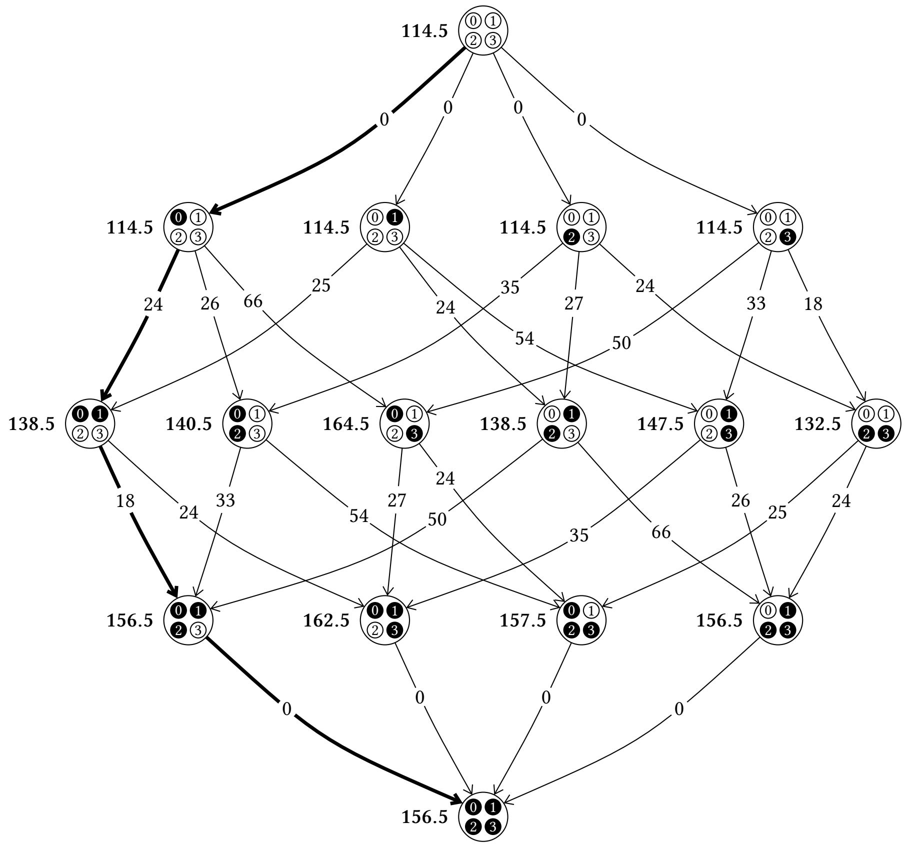


Figure 6.2: Exact DD for the SRFLP instance given by Figure 6.1. Free and fixed departments are respectively represented by empty and filled circles.

This formula immediately follows from Equation (6.1) since $\bar{s}^j$ – the complement of s^j – contains fixed departments placed before position j and $s^j \setminus \{x_j\}$ contains free departments, which will be placed after position j.

- The root value is $v_r = K$ from Equation (6.1).

Example 6.4.1. *Figure 6.2 shows the exact DD that can be obtained for the instance described by Figure 6.1 and with the DP model defined in this section. Let us detail the computation of the transition value between the nodes associated with states* $\{2,3\}$ *and* $\{3\}$: $h_2(\{2,3\},2) = L_2 \sum_{i\in\{0,1\}} \sum_{k\in\{2,3\}\setminus\{2\}} C_{ik} = L_2 \times (C_{03} + C_{13}) = 2 \times (5+4) = 18$. *The permutation depicted in Figure 6.1 indeed corresponds to the longest path in the exact DD and that its value is equal to the one computed in Example 6.2.1.*

Speeding up the computation of transition costs We also store in the states an array containing the *cut values* of each free department: the sum of all traffic intensities from the fixed departments and each free department. It allows to reduce the computational complexity of the transition costs from $O\left(n^2\right)$ to $O(n)$ and will also be useful when designing a lower bound, as explained in Section 6.4.4. For a state s^j and each department $i \in N$, we define:

$$s^j_{cut}[i] = \begin{cases} \sum_{j\in\bar{s}^j} C_{ij}, & \text{if } i \in s^j \\ 0, & \text{otherwise.} \end{cases} \tag{6.18}$$

These cut values can be updated in $O(n)$ during a transition $t_j\left(s^j, x_j\right)$ by applying:

$$s^{j+1}_{cut}[i] = \begin{cases} s^j_{cut}[i] + C_{ix_j}, & \text{if } i \in s^j \setminus \{x_j\} \\ 0, & \text{otherwise.} \end{cases}$$

This allows redefining the transition value functions as:

$$h_j\left(s^j, x_j\right) = \begin{cases} L_{x_j} \sum_{i\in s^j\setminus\{x_j\}} s^j_{cut}[i], & \text{if } x_j \in s^j \\ 0, & \text{otherwise,} \end{cases}$$

with the cut values of the root state initialized at zero: $\hat{r}_{cut} = 0^n$.

Example 6.4.2. *Considering the instance of Figure 6.1, we compute the cut values for the state* $s = \{2, 3\}$. *We have that* $s_{cut}[0] = s_{cut}[1] = 0$ *since departments 0 and 1 are already placed. For the free departments, we apply Equation* (6.18) *and obtain:* $s_{cut}[2] = C_{02}+C_{12} = 3+1 = 4$ *and* $s_{cut}[3] = C_{03}+C_{13} = 5+4 = 9$. *The computation made in Example 6.4.1 becomes:* $h_2(\langle\{2,3\}, cut = (0,0,4,9)\rangle, 2) = L_2 \sum_{i\in\{2,3\}\setminus\{2\}} cut_i = L_2 \times cut_3 = 2 \times 9 = 18$.

6.4.2 cSRFLP

Adding constraints to the DP model is done through the predicates $valid_j : S_j \times D_j \longrightarrow \{true, false\}$ for $j = 0, \dots, n-1$. They are used in the transition functions to filter out infeasible solutions:

$$t_j\left(s^j, x_j\right) = \begin{cases} s^j \setminus \{x_j\}, & \text{if } x_j \in s^j \wedge valid_j(s^j, x_j) \\ \hat{0}, & \text{otherwise.} \end{cases}$$

For clarity, we split the predicates $valid_j$ into several conditions, corresponding each to a specific constraint:

$$valid_j(s^j, x_j) = p_j(s^j, x_j) \wedge o_j(s^j, x_j) \wedge r_j(s^j, x_j)$$

with p_j, o_j and r_j concerning respectively positioning, ordering and relation constraints:

$$\begin{aligned} p_j(s^j, x_j) =& (position(x_j) = \perp \wedge department(j) = \perp) \\ & \vee position(x_j) = j \end{aligned} \tag{6.19}$$

$$o_j(s^j, x_j) = predecessors(x_j) \subseteq \bar{s}^j \tag{6.20}$$

$$\begin{aligned} r_j(s^j, x_j) =& (previous(x_j) = 0 \wedge \nexists k \in s^j : previous(k) \in \bar{s}^j) \\ & \vee previous(x_j) \in \bar{s}^j. \end{aligned} \tag{6.21}$$

In Equation (6.19), p_j checks that either department x_j and position j are both unconstrained, or that department x_j is constrained to be at position j. For ordering constraints, Equation (6.20) verifies that all predecessors of department x_j have already been placed. The predicates r_j for relation constraints are slightly more complicated. Either x_j has no relation constraint, then it can only be placed if no other free department has a relation constraint with a fixed department, or x_j has a relation constraint and $previous(x_j)$ must be a fixed department.

Example 6.4.3. *Figure 6.3 shows exact DDs obtained with this extended DP model for the same instance as for Figure 6.2, but with additional constraints.*

(a) We add the following positioning constraint: $position(2) = 2$. As one can verify in Figure 6.3(a), transitions concerning department 2 are only performed between nodes of L_2 and L_3. Moreover, these are the only transitions occurring between those layers since placing any other department in this position would violate the constraint.

(b) On top of the positioning constraint $position(2) = 2$, an ordering constraint is added: $predecessors(3) = \{0\}$. As a result, transitions from the root node of Figure 6.3(b) do not involve department 3 since department 0 has not been placed yet.

In both cases, we can observe that the only effect of adding constraints to the problem is to remove arcs in the DDs. It generates smaller DDs and thus simplifies the resolution of the problem.

6.4.3 Relaxation

It is quite straightforward to specify relaxation operators for the cSRFLP. Indeed, one can extend the DP state to contain a set of free departments that are free in all states to merge, and a set of *possibly* free departments that are free in *some* states to merge, as done with the remaining locations in TSPTW formulation presented in Section 3.4.1. However, we will not detail such a

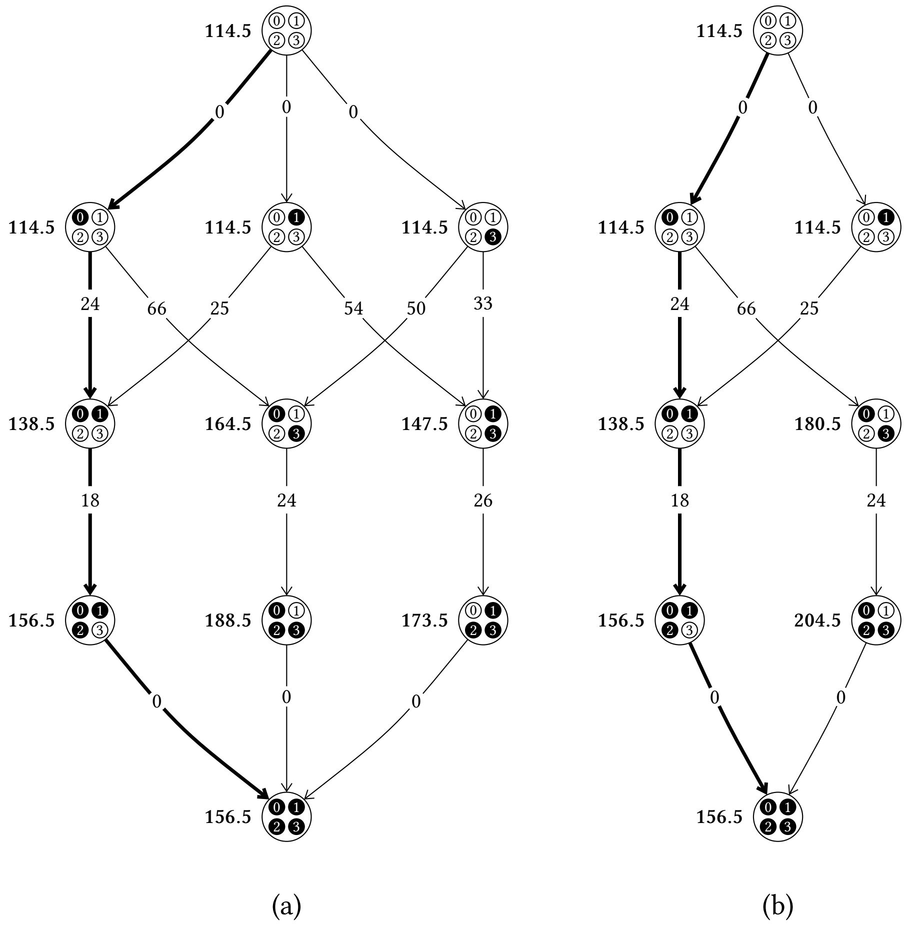


Figure 6.3: Exact DDs for the SRFLP instance given by Figure 6.1 and with the additional constraints given in Example 6.4.3.

relaxation here because it unnecessarily complicates the explanations of the DP model and the RLB, while ultimately the restricted B&B variant performs best for this DP model without involving the state merging relaxation.

6.4.4 Rough Lower Bound

In order to derive the RLB from a node u, the next theorem shows that the cost to optimally complete the partial solution of node u can be decomposed in two terms: one solely involving the free departments and the other one involving the cost between free and fixed departments.

Theorem 6.4.1. *Given a node u and its state $\sigma(u) = s$, let $\pi^*|_u$ be the best* *ordering one can obtain when crossing node u. For conciseness, we set $\pi = \pi^*|_u$. We have the equivalence:*

$$SRFLP(\pi) - v^*(u) = \underbrace{\sum_{i \in s} \sum_{\substack{j \in s \\ \pi(i)<\pi(j)}} C_{ij} \sum_{\substack{k \in s \\ \pi(i)<\pi(k)<\pi(j)}} L_k}_{\textit{free departments layout cost}} + \underbrace{\sum_{j \in s} s_{cut}[j] \sum_{\substack{k \in s \\ \pi(k)<\pi(j)}} L_k}_{\textit{cost w.r.t. fixed departments}} .$$

Proof.

$$\begin{aligned}
\Delta &= SRFLP(\pi) - v^*(u) \\
&= \sum_{k=0}^{n-1} L_k \sum_{\substack{i=0 \\ \pi(i)<\pi(k)}}^{n-1} \sum_{\substack{j=0 \\ \pi(k)<\pi(j)}}^{n-1} C_{ij} + K - \left(\sum_{\substack{k=0 \\ \pi(k)\le|\bar{s}|}}^{n-1} L_k \sum_{\substack{i=0 \\ \pi(i)<\pi(k)}}^{n-1} \sum_{\substack{j=0 \\ \pi(k)<\pi(j)}}^{n-1} C_{ij} + K \right) \\
&= \sum_{\substack{k=0 \\ \pi(k)>|\bar{s}|}}^{n-1} L_k \sum_{\substack{i=0 \\ \pi(i)<\pi(k)}}^{n-1} \sum_{\substack{j=0 \\ \pi(k)<\pi(j)}}^{n-1} C_{ij} \\
&= \sum_{\substack{k=0 \\ \pi(k)>|\bar{s}|}}^{n-1} L_k \left(\sum_{\substack{i=0 \\ \pi(i)\le|s|}}^{n-1} \sum_{\substack{j=0 \\ \pi(k)<\pi(j)}}^{n-1} C_{ij} + \sum_{\substack{i=0 \\ |s|<\pi(i)<\pi(k)}}^{n-1} \sum_{\substack{j=0 \\ \pi(k)<\pi(j)}}^{n-1} C_{ij} \right) \\
&= \sum_{k \in s} L_k \left(\sum_{i \in \bar{s}} \sum_{\substack{j \in s \\ \pi(k)<\pi(j)}} C_{ij} + \sum_{\substack{i \in s \\ \pi(i)<\pi(k)}} \sum_{\substack{j \in s \\ \pi(k)<\pi(j)}} C_{ij} \right) \\
&= \sum_{k \in s} L_k \sum_{i \in \bar{s}} \sum_{\substack{j \in s \\ \pi(k)<\pi(j)}} C_{ij} + \sum_{k \in s} L_k \sum_{\substack{i \in s \\ \pi(i)<\pi(k)}} \sum_{\substack{j \in s \\ \pi(k)<\pi(j)}} C_{ij} \\
&= \sum_{k \in s} L_k \sum_{\substack{j \in s \\ \pi(k)<\pi(j)}} \sum_{i \in \bar{s}} C_{ij} + \sum_{i \in s} \sum_{\substack{j \in s \\ \pi(i)<\pi(j)}} C_{ij} \sum_{\substack{k \in s \\ \pi(i)<\pi(k)<\pi(j)}} L_k \\
&= \sum_{k \in s} L_k \sum_{\substack{j \in s \\ \pi(k)<\pi(j)}} s_{cut}[j] + \sum_{i \in s} \sum_{\substack{j \in s \\ \pi(i)<\pi(j)}} C_{ij} \sum_{\substack{k \in s \\ \pi(i)<\pi(k)<\pi(j)}} L_k \\
&= \sum_{j \in s} s_{cut}[j] \sum_{\substack{k \in s \\ \pi(k)<\pi(j)}} L_k + \sum_{i \in s} \sum_{\substack{j \in s \\ \pi(i)<\pi(j)}} C_{ij} \sum_{\substack{k \in s \\ \pi(i)<\pi(k)<\pi(j)}} L_k
\end{aligned}$$

□

Those two terms of the equivalence in Theorem 6.4.1 cannot be evaluated exactly in a cheap way as this would be as difficult as solving the original problem. Nonetheless, one can compute an efficient lower bound for each term independently. For a state s, the value of the RLB is given by:

$$\underline{v}_{rlb}(s) = LB_{edge}(s) + LB_{cut}(s)$$

where $LB_{edge}(s)$ is a lower bound on the free departments layout cost and $LB_{cut}(s)$ is a lower bound on the cost induced by the cut values of free departments.

6.4.4.1 Free departments layout cost

The first lower bound LB_{edge} is an under-approximation of the internal layout cost of free departments. Given a subset of departments, we compute a lower bound on the cost of its optimal layout by multiplying each pairwise traffic intensity by an optimistic distance. If we must place n departments on a line, $n - k$ pairs of departments will have $k - 1$ departments between them (see Figure 6.1). In order to under-approximate the layout cost, we greedily multiply the highest traffic intensities by the smallest distance possible. Since we cannot assume any particular ordering of the free departments, the distances between pairs of free departments are unknown. Still, we can compute lower bounds on those distances if we sort the free departments by increasing length and assume that a separation of k departments will be formed by the k shortest departments. This lower bound can be seen as a generalization of the Edges method [Pet03] designed for the MinLA.

In practice, a list containing all pairwise traffic intensities in decreasing weight order is precomputed, as stated by the precondition of Algorithm 16. The same is done for the department lengths. We then only need to traverse those lists and multiply each traffic intensity value by the adequate cumulative length. The complexity of the algorithm is $O\left(n^2\right)$ since there are $\frac{n(n-1)}{2}$ pairs in total.

Example 6.4.4. *Let us illustrate the computation of this lower bound on the root state of the DD in Figure 6.2. We first create the list of traffic intensities sorted decreasingly: edge* $= [C_{01} = 8, C_{23} = 6, C_{03} = 5, C_{13} = 4, C_{02} = 3, C_{12} = 1]$ *and the list of free department lengths sorted increasingly: length* $= [L_2 = 2, L_1 = 3, L_0 = 5, L_3 = 6]$. *There are 3 pairs of departments with 0 departments in between, 2 pairs with 1 department in between and 1 pair with 2 departments in between.*

$$\begin{aligned} LB_{edge}(\hat{r}) &= 0 \times C_{01} + 0 \times C_{23} + 0 \times C_{03} + L_2 C_{13} + L_2 C_{02} + (L_2 + L_1) C_{12} \\ &= 0 \times 8 + 0 \times 6 + 0 \times 5 + 2 \times 4 + 2 \times 3 + (2 + 3) \times 1 = 19 \end{aligned}$$

Algorithm 16 Computation of $LB_{edge}(s)$.

```
Require: $edge = sorted_{\geq}\left(\{\langle c : C_{ij}, dep1 : i, dep2 : j\rangle \mid 1 \leq i < j \leq n\}\right)$
    and $length = sorted_{\leq}\left(\{\langle l : L_i, dep : i\rangle \mid 1 \leq i \leq n\}\right)$
 1: $lb \leftarrow 0, cumul\_l \leftarrow 0, i \leftarrow 1, j \leftarrow 1$
 2: for $k \leftarrow 1$ to $|s| - 1$ do
 3:     for $l \leftarrow 1$ to $k$ do
 4:         while $edge[i].dep1 \notin s \vee edge[i].dep2 \notin s$ do
 5:             $i \leftarrow i + 1$
 6:         $lb \leftarrow lb + cumul\_l \times edge[i].c$
 7:         $i \leftarrow i + 1$
 8:     while $length[j].dep \notin s$ do
 9:         $j \leftarrow j + 1$
10:     $cumul\_l \leftarrow cumul\_l + length[j].l$
11:     $j \leftarrow j + 1$
12: return $lb$
```

6.4.4.2 Cost with respect to fixed departments

The second term of the RLB is related to the cut values of free departments and a lower bound is given by the *first-generation bound* described in [Sim69]. Given a department i placed first on the line, the minimum total cost with respect to i is defined as:

$$MTC(i) = \min_{\pi} \sum_{\substack{j=1 \\ i \neq j}}^{n} C_{ij} \sum_{\substack{k=1 \\ \pi(k) < \pi(j)}}^{n} L_k \tag{6.22}$$

and Lemma 6.4.2 tells us how to find the optimal arrangement π.

Lemma 6.4.2. *Suppose that department i is placed in first position on the line. For every other department j compute the cost-to-length ratio $r_j = \frac{C_{ij}}{L_j}$. The optimal arrangement, which yields $MTC(i)$ is obtained by ordering the departments according to decreasing values of this ratio r_j, the department with the greatest r_j being adjacent to i.*

This lower bound can also be used when several departments are placed in the leftmost positions on the line. We only need to consider all fixed departments as a single department connected to free departments with traffic intensities given by the respective cut values, exactly as in the second term of the equivalence given by Theorem 6.4.1. As the free departments need to be sorted by decreasing cut-to-length ratios, the time complexity of this lower bound is $O(n \log(n))$.

Example 6.4.5. *We compute the lower bound for the state $s = \{1, 2, 3\}$ with $s_{cut} = (0, 8, 3, 5)$. The departments are first sorted as follows:*

$$order = \left[\frac{s_{cut}[1]}{L_1} = \frac{8}{3}, \frac{s_{cut}[2]}{L_2} = \frac{3}{2}, \frac{s_{cut}[3]}{L_3} = \frac{5}{6}\right].$$

We then compute the lower bound as the total cost with respect to all fixed departments:

$$\begin{aligned} LB_{cut}(s) &= 0 \times s_{cut}[1] + L_1 s_{cut}[2] + (L_1 + L_2) s_{cut}[3] \\ &= 0 \times 8 + 3 \times 3 + (3+2) \times 5 = 34. \end{aligned}$$

6.5 Computational Experiments

In this section, we draw a comparison between the existing techniques to solve the cSRFLP to optimality. Namely, the MIP model from [Liu+21], the MIP model introduced in [Ama09] and extended in Section 6.3 and the DD-based approach presented throughout the rest of the chapter. In the following, they are respectively referred to as Liu, Amaral and DDO. The MIP models were implemented and evaluated using Gurobi version 9.5.2 [Gur22]. The DD-based approach was implemented with DDO and evaluated with various configurations discussed in the previous chapters. DDs with a fixed maximum width $W = n$ were used, with n the number of variables in the problem instance.

The instances used in the experiments are classical SRFLP instances taken from [Ama06; Ama08; AV08; HK91; Sim69] with up to 25 departments. We then created admissible sets of constraints for each problem size:

- constraint sets with 2, 4, 6, 8 and 10 positioning, ordering or relation constraints.
- constraint sets with 0, 2, 4, 6, 8 and 10 constraints of each type.

For each of these scenarios, 3 different random sets of constraints were generated, except for the case with no constraints. Note that an instance with n departments can not have more than n positioning constraints, and that similar limits exist for the other types of constraints, we thus have up to 63 sets of constraints for each problem size. The three models were applied to all combinations of instances and constraints with a time limit of 600 seconds for each.

6.5.1 Number of Benchmark Instances Solved

Figure 6.4 shows the cumulative number of instances solved by each algorithm over time, for instances where either all types of constraints were applied, or a single one of them. Our first observation is that the two models presented in this chapter clearly outperform the one from [Liu+21], which fails to solve most of the instances under the time limit regardless of the type of constraints applied. We can also notice that the caching mechanism

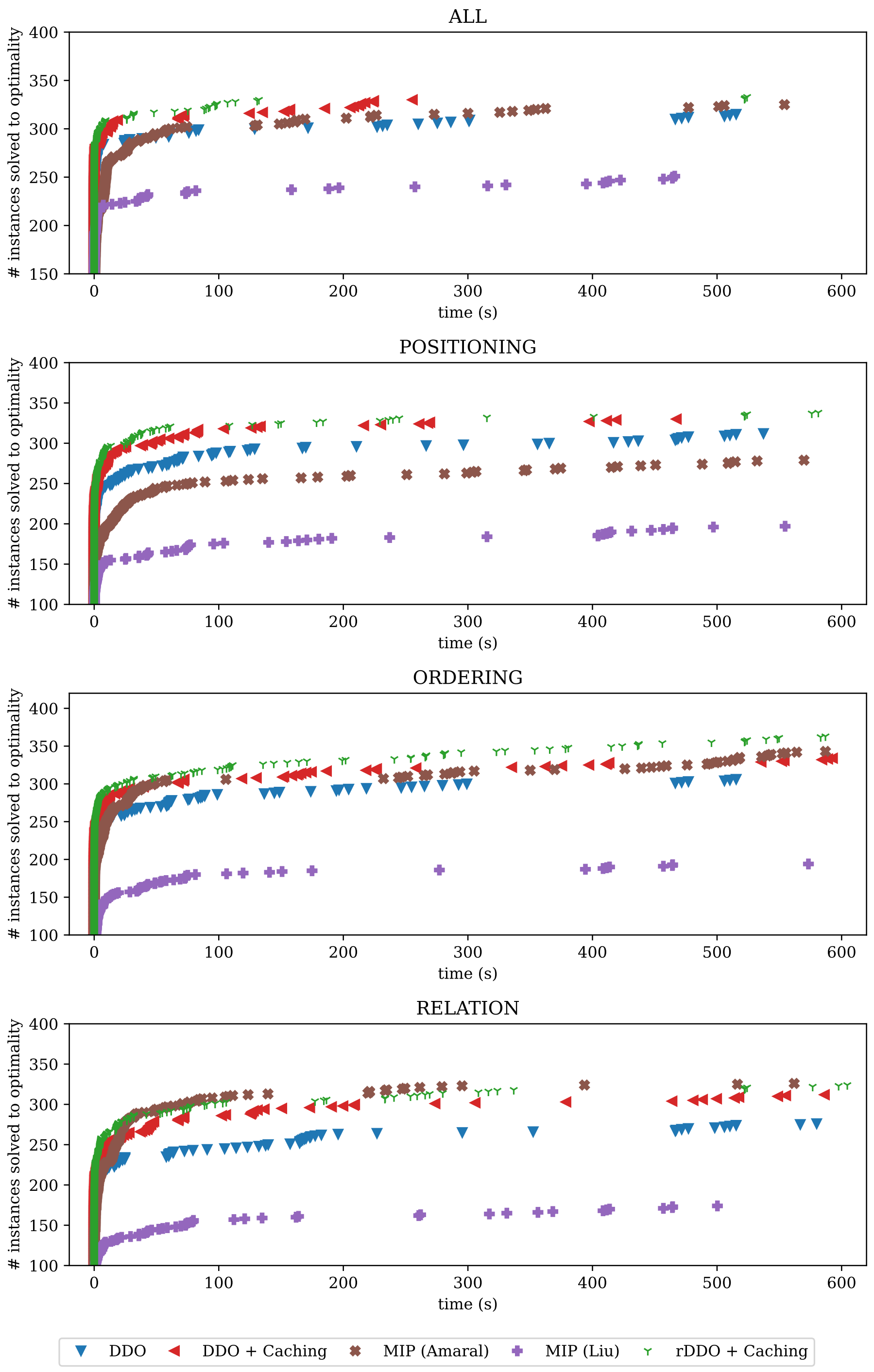


Figure 6.4: Number of instances solved by each algorithm for the different types of constraints.

presented in Chapter 4 is once again beneficial for the DD-based approach. However, the best performing configuration of DDO is by a small margin its restricted version denoted rDDO+C, showing that the state merging-based relaxation for the cSRFLP does not necessarily generate a lot of additional pruning when comparing to the RLB. The comparison between rDDO+C and Amaral differs from one type of constraint to the other. Indeed, rDDO+C has a slight advantage when considering all types of constraints at once and for ordering constraints, solving respectively 7 and 20 additional instances. For instances with positioning constraints, all DD-based approaches perform better than Amaral. In particular, rDDO+C manages to solve 338 instances whereas Amaral could only solve 279 of them. However, Amaral is able to solve two more instances than rDDO+C when facing relation constraints.

6.5.2 Evolution of Performance with the Number of Constraints

We further discuss how the performance of rDDO+C and Amaral is impacted by the addition of the different types of constraints by representing the solving times for each type and number of constraints imposed on Figure 6.5. To get a sense of this evolution of the performance, we restrict the results cSRFLP instances created from SRFLP instances that can be solved by both algorithms when no constraints are considered. Regardless of the type of constraints involved, we notice that the more constraints we add, the faster the DD-based approach gets. The constraints in the DD formulation are indeed handled very efficiently because all infeasible solutions are automatically pruned in the transition functions, which results in a smaller DP graph to explore. The same cannot be said about Amaral, since some instances with between 2 and 8 positioning constraints take more time to solve than their corresponding unconstrained instance. This is probably caused by positioning constraints being modeled with a sum of n variables on the left side of an equality – see Equations (6.11) and (6.12). On the contrary, ordering and relation constraints are modeled very naturally in Amaral because it uses relative ordering variables. Adding these types of constraints thus tightens the model and generally reduces the execution time. This is especially true for relation constraints, as they force many assignments with Equation (6.17).

6.5.3 End Gaps

Finally, we compare the end optimality gap reached by the algorithms and configurations studied in this chapter on Figure 6.6. For the hardest instances, DDO+C obtains the smallest optimality gap even though it solved fewer instances than rDDO+C. This suggests that the lower bounds derived from relaxed DDs can be tighter than those obtained from the RLB, and yet they do not speed up the algorithm, probably because they are more demanding to

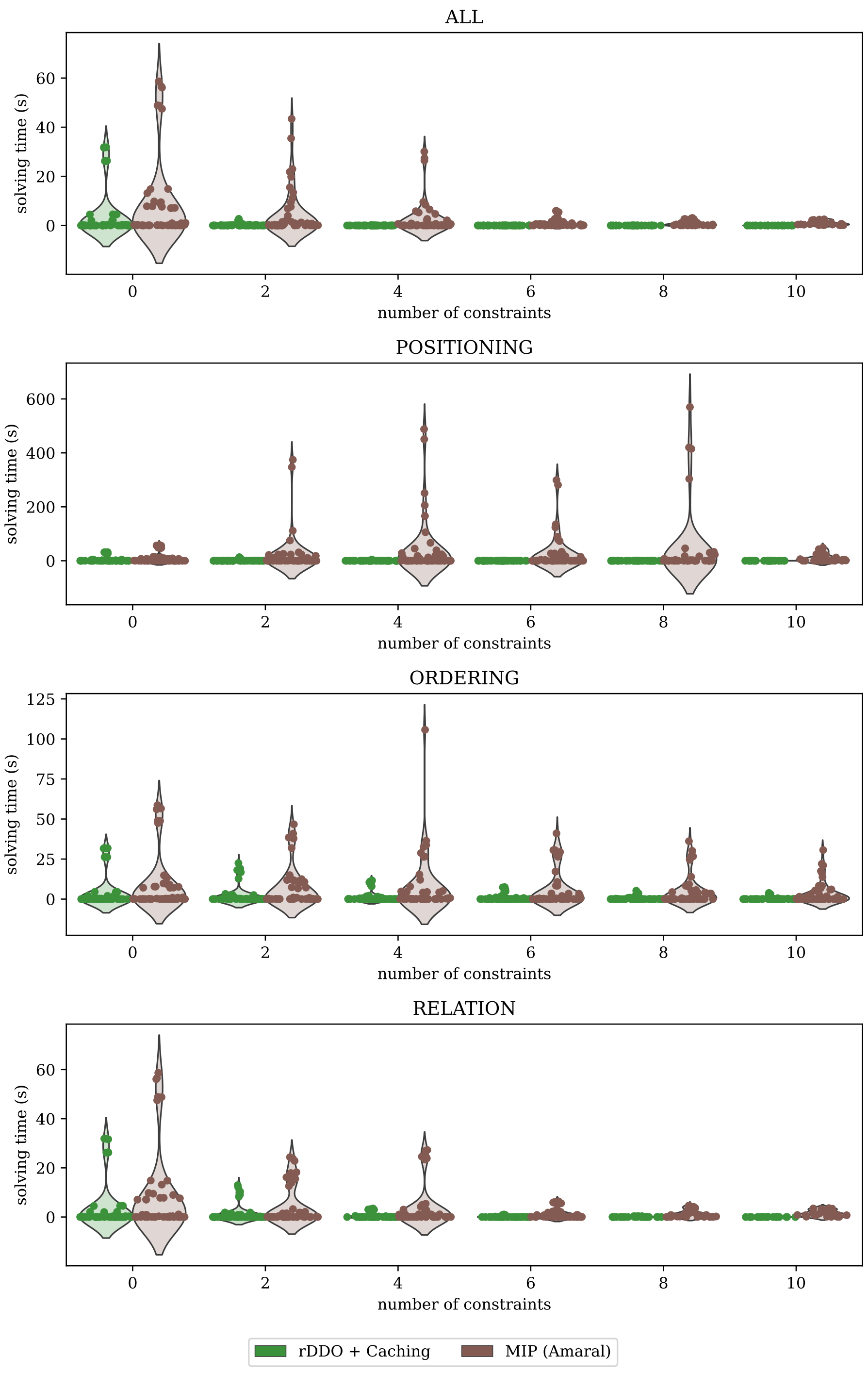


Figure 6.5: Solving time of instances solved by both Amaral and rDDO+C for the different types of constraints, with respect to the number of constraints of each type.

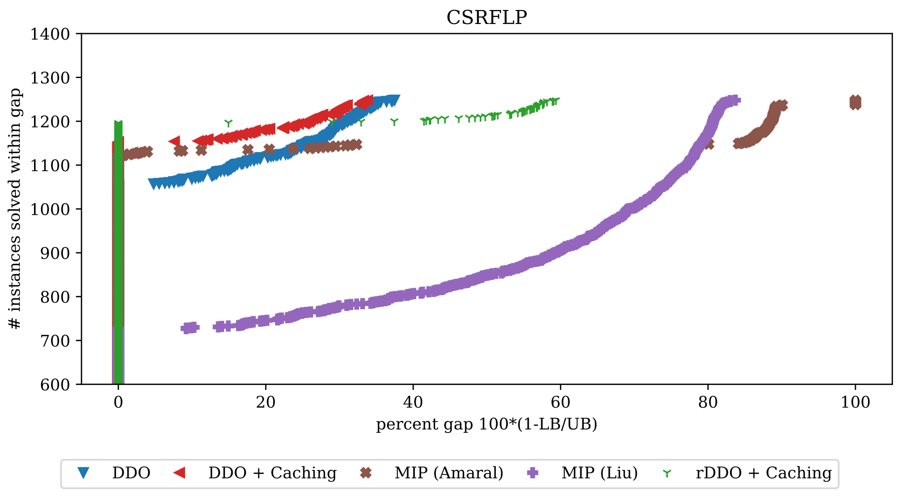


Figure 6.6: Number of instances solved by all algorithms and configurations within a given optimality gap.

compute. The end optimality gaps achieved by Amaral are quite poor compared to the DD-based configurations, with 101 of the 129 unsolved instances having an optimality gap above 80%. For those instances, even Liu reaches a tighter end gap even though it could not solve most of the instances.

6.6 Conclusion

In this chapter, two novel exact models for the cSRFLP have been presented: an extension of the MIP model from [Ama09] for the SRFLP and a DD-based approach starting from the DP model of [PQ81]. The additional positioning, ordering and relation constraints have been integrated in both approaches and an RLB was presented for the DD-based one. The computational experiments have shown that these two new formulations perform significantly better than the only MIP model previously introduced in the literature, and for all types of constraints considered. Both models introduced have their strengths, the DD-based approach incorporates the three types of constraints very efficiently, especially positioning constraints. On the other hand, the MIP model integrates ordering and relation constraints very well and can be easily implemented with any MIP solver. The optimality gap analysis shows that the bounds obtained within the DD-based approach are tighter than those of Amaral. Therefore, coupling both formulations could constitute a promising direction for further research.

Conclusion | 7

7.1 Summary of the Contributions

In this thesis, we proposed – or showed how to integrate – several additional filtering mechanisms for the DD-based B&B algorithm, after recalling how it operates in Chapter 2. The first one was discussed in Chapter 3 and consisted in incorporating dominance rules in the solver to efficiently detect and discard redundant parts of the search space, based on problem-specific information provided through a few simple dominance operators. For the TSPTW, the ALP, the LCS and the KP, the positive impact of dominance rules was clearly observed experimentally in terms of number of node expansions required to solve difficult instances, resulting in shorter execution times without necessarily increasing the memory consumption of the solver. The ability of the solver to quickly find good solutions was also improved, especially for highly constrained problems.

Chapter 4 presented a second pruning ingredient based on the caching of expansion thresholds for DP states visited by the approximate DDs. Its application does not require any additional problem-specific modeling, as it exploits the typical structure of DP state graphs. Furthermore, it builds upon the RUBs and the LocBs introduced in [Gil+21] as well as the dominance rules. Our experimental evaluation was pursued with two additional optimization problems – the PSP and the TalentSched – and confirmed our intuition that many DP states were unnecessarily visited multiple times. Mitigating this issue with the proposed caching strategy again considerably reduced the number of nodes expanded by the algorithm, and, consequently, also the solving times. Interestingly, the combination of the dominance detection procedures with the caching component yielded significantly better results than both improvements alone, except for the KP for which dominance rules were less impactful. However, the memory footprint of maintaining the collection of expansion thresholds was found to be heavier than the one induced by dominance rules, although not when used together.

Next, we explained in Chapter 5 how aggregate dynamic programming could be leveraged to construct node selection heuristics and dual bounds for approximate DDs. For the ALP, the PSP and the TalentSched, the former were shown to help guide restricted DDs by causing the algorithm to produce better first solutions in a majority of the cases, while the complementary filtering generated by the latter resulted in more instances being solved. With both aggregation-based ingredients combined, even more benchmark instances could be solved and the optimality gap for unsolved instances was frequently tightened. The validation of those components was further supported by experimenting with a restricted B&B that does not rely on relaxed DDs to compute bounds, which achieved similar or even better performance than the best classical B&B configurations.

Finally, Chapter 6 concerned the application of the DD-based optimization framework to a constrained facility layout problem, and its comparison with a MIP formulation. It showcased the ability of the DD-based approach to easily integrate additional constraints, both in terms of modeling and resolution. Indeed, the constraints are handled by discarding some transitions and result in a smaller DP state space to explore, whereas they sometimes complicate the solving process for MIP.

To conclude this summary of our contributions, we would like to stress that all the proposed filtering mechanisms are compatible with each other, and that they only reinforce the relevance of those previously developed such as the RUBs and the LocBs. Moreover, the implementation of the techniques introduced in Chapters 3 and 4 are now integral parts of the latest release of the DDO solver. We hope that these theoretical and software contributions will participate in the success and adoption of the DD-based optimization framework, and possibly pave the way for future findings.

7.2 Perspectives

This section reflects on the contributions of this thesis and the questions that remain open, and suggests some directions for future research in the field.

7.2.1 Reducing Memory Consumption

As discussed in Chapter 3, detecting dominance relations between nodes belonging to DDs compiled at different stages of the B&B algorithm requires maintaining a collection of non-dominated node utilities in memory. Similarly, the caching strategy introduced in Chapter 4 involves memorizing an expansion threshold for each DP state reached by an exact node of any relaxed DD. In the worst case, the algorithm might end up storing a utility or an expansion threshold for every possible DP state of the model. To mitigate the memory footprint of these filtering techniques, one possibility is to bound the size of the associated data structures, i.e. the *Fronts* and the *Cache*. However, this raises several questions:

- Are there additional criteria that can determine whether a utility or a dominance threshold has become obsolete? We explained that expansion thresholds above the first active layer could safely be removed. Furthermore, we can delete expansion thresholds associated with states that are known to be dominated. Could we also detect cases where the expansion threshold of a DP state is equivalent to the expansion threshold stored for all its parent states, and thus discard it?

- What kind of eviction policies do we use to decide which utilities or thresholds to withdraw? Are those concerning states at early stages of the DP model more important? Is it relevant to use metrics such as the number of successful pruning operations generated since they were introduced?

- Is there an ideal maximum number of stored entries that best balances memory consumption and filtering strength?

Note that we can probably use a single data structure to store both types of information. However, in the current implementation, both mechanisms are kept separate for clarity and to allow each component to be easily enabled or disabled.

Besides these considerations regarding the contributions of this thesis, it is worth mentioning that the original B&B algorithm also suffers from worst-case space complexity equivalent to storing the complete DP state graph, as mentioned in Section 4.4. Given that the caching mechanism introduced in Chapter 4 would still prevent the algorithm from unnecessarily expanding previously visited DP states, we wonder how a *depth-first* search at the upper level of the B&B algorithm would perform. It would remove the need for a *Fringe* and thus clearly alleviate the memory requirements of the algorithm. Combined with bounded-size *Fronts* and *Cache*, it could possibly result in an algorithm that can be effectively bounded in memory.

7.2.2 Alternative Relaxation Schemes

The state merging-based relaxation is a key component of the original DD-based B&B algorithm, and proved to provide tight bounds for multiple classes of problem. However, the computational experiments of Chapters 5 and 6 showed that relaxed DDs can fail to generate significantly more pruning than RUBs for some DP models, e.g. the TalentSched and the cSRFLP. Previous work has successfully investigated the improvement of the relaxed DD bounds by applying the state merging to groups of nodes that share similar state information, as well as strengthening path values by incorporating penalties for assignments that violate some constraints, obtained through Lagrangian relaxation [BCH15; Hoo19]. Still, there might exist other generic relaxation schemes that would be better suited to some specific problems.

Chapter 5 proposed one such alternative that relies on aggregate dynamic programming and obtained good results, especially on the PSP and the ALP, even when used in isolation in the restricted B&B variant. While it is a promising research direction, this relaxation scheme is not applicable to any given problem, it requires advanced additional modeling, and it may yield poor bounds if the problem data is not suitable for aggregation at all.

Both the state merging and the aggregation-based relaxation schemes are closely related to various types of *abstraction heuristics* introduced in the classical planning literature [HHH+07; Lar+10; Hel+14; SWH14; SH18]. Therefore, it would be very interesting to clarify their connections, and possibly transfer some ideas to the field of DD-based optimization. Note that [KB23] already applied classical search algorithms to combinatorial optimization problems, which also calls for unifying the research efforts of these two communities.

7.2.3 Integration in CP Solvers

Although DD-based optimization is a very efficient technique to solve large DP models, it is a real challenge to make it widely adopted, as it constitutes one more technology for practitioners to learn, and with a modeling style that is very different from traditional approaches such as MIP and CP. Recently, the Haddock modeling language and system [GMH20; GMH22; GMH23] translated most of the DD-based optimization components inside a CP solver, with DDs acting as constraint stores and propagators. Therefore, it can replicate the behavior of a DD-based B&B solver while providing a CP-like modeling interface, which is a very promising direction to make the power of DD-based optimization available to a larger audience. We believe that some of the contributions of this thesis can be directly integrated into such system, and could have an impact as significant as that shown in this thesis for the pure DD-based B&B solver. The dominance rules and the caching strategy are good candidates for this, although it is unclear whether they can be used as is in the case of external constraints that are not imposed on the DD. Indeed, if external constraints invalidate nodes that have been used to derive a dominance relation or to compute an expansion threshold, the associated filtering procedures might become incorrect.

7.2.4 Learning Node Selection Heuristics

In [Cap+22], reinforcement learning was used to automatically discover good variable orderings and, in doing so, improve the bounds obtained with approximate DDs. Another opportunity for integrating learning into the DD-based optimization framework concerns the node selection heuristics used to compile restricted DDs. Indeed, it was shown in Chapter 5 that node selection heuristics infused with knowledge about the global problem structure can have great impact on the quality of the solutions found, which in turn can speed up the B&B algorithm. Moreover, a node selection heuristic can be constructed from any evaluation function that estimates the potential of a node to lead to the optimal solution, using it to sort the nodes of a layer from most to least promising and retaining the W most promising ones. Given the usually small amount of information contained in DP states and the latest advances in the field of machine learning, the task of learning such evaluation function seems approachable. This could also be applied to the selection of the B&B node to explore next, as done for MIP in [HDE14; LCL22].

Bibliography

[Ake78] S. B. Akers. “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* 27.06 (1978), pp. 509–516.

[AKV05] M. F. Anjos, A. Kennings, and A. Vannelli. “A semidefinite optimization approach for the single-row layout problem with unequal dimensions”. In: *Discrete Optimization* 2.2 (2005), pp. 113–122.

[AL13] A. R. S. Amaral and A. N. Letchford. “A polyhedral approach to the single row facility layout problem”. In: *Mathematical programming* 141.1-2 (2013), pp. 453–477.

[Ama06] A. R. Amaral. “On the exact solution of a facility layout problem”. In: *European Journal of operational research* 173.2 (2006), pp. 508–518.

[Ama08] A. R. Amaral. “An exact approach to the one-dimensional facility layout problem”. In: *Operations Research* 56.4 (2008), pp. 1026–1033.

[Ama09] A. R. Amaral. “A new lower bound for the single row facility layout problem”. In: *Discrete Applied Mathematics* 157.1 (2009), pp. 183–190.

[And+07] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. “A constraint store based on multivalued decision diagrams”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 118–132.

[Asc96] N. Ascheuer. “Hamiltonian path problems in the on-line optimization of flexible manufacturing systems”. PhD thesis. University of Technology Berlin, 1996.

[AV08] M. F. Anjos and A. Vannelli. “Computing globally optimal solutions for single-row layout problems using semidefinite programming and cutting planes”. In: *INFORMS Journal on Computing* 20.4 (2008), pp. 611–617.

[Axs83] S. Axsäter. “State aggregation in dynamic programming—An application to scheduling of independent jobs on parallel processors”. In: *Operations Research Letters* 2.4 (1983), pp. 171–176.

[AY09] M. F. Anjos and G. Yen. "Provably near-optimal solutions for very large single-row facility layout problems". In: *Optimization Methods & Software* 24.4-5 (2009), pp. 805–817.

[BB07] C. Blum and M. J. Blesa. "Probabilistic beam search for the longest common subsequence problem". In: *International Workshop on Engineering Stochastic Local Search Algorithms*. Springer. 2007, pp. 150–161.

[BBS87] J. C. Bean, J. R. Birge, and R. L. Smith. "Aggregation in dynamic programming". In: *Operations Research* 35.2 (1987), pp. 215–220.

[BCH15] D. Bergman, A. A. Cire, and W.-J. van Hoeve. "Lagrangian bounds from decision diagrams". In: *Constraints* 20 (2015), pp. 346–361.

[Bec+05] B. Becker, M. Behle, F. Eisenbrand, and R. Wimmer. "BDDs in a branch and cut framework". In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2005, pp. 452–463.

[Bel54] R. Bellman. "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6 (Nov. 1954), pp. 503–515.

[Ber+12] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. 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*. Springer. 2012, pp. 34–49.

[Ber+14a] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. "Optimization bounds from binary decision diagrams". In: *INFORMS Journal on Computing* 26.2 (2014), pp. 253–268.

[Ber+14b] D. Bergman, A. A. Cire, W.-J. van Hoeve, and T. Yunes. "BDD-based heuristics for binary optimization". In: *Journal of Heuristics* 20.2 (2014), pp. 211–234.

[Ber+16] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. "Discrete optimization with decision diagrams". In: *INFORMS Journal on Computing* 28.1 (2016), pp. 47–66.

[BF16] C. Blum and P. Festa. "Longest common subsequence problems". In: *Metaheuristics for String Problems in Bioinformatics* (2016), pp. 45–60.

[BMR93] L. Bianco, A. Mingozzi, and S. Ricciardelli. "The traveling salesman problem with cumulative costs". In: *Networks* 23.2 (1993), pp. 81–91.

[Bry86] R. E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[Cap+22] Q. Cappart, D. Bergman, L.-M. Rousseau, I. Prémont-Schwarz, and A. Parjadis. “Improving Variable Orderings of Approximate Decision Diagrams Using Reinforcement Learning”. In: *INFORMS Journal on Computing* 34.5 (2022), pp. 2552–2570.

[CCB22] M. P. Castro, A. A. Cire, and J. C. Beck. “Decision diagrams for discrete optimization: A survey of recent advances”. In: *INFORMS Journal on Computing* 34.4 (2022), pp. 2271–2295.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. “Model checking and abstraction”. In: *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1512–1542.

[CGS22] V. Coppé, X. Gillard, and P. Schaus. “Solving the Constrained Single-Row Facility Layout Problem with Decision Diagrams”. In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[CGS23a] V. Coppé, X. Gillard, and P. Schaus. “Boosting Decision Diagram-Based Branch-And-Bound by Pre-Solving with Aggregate Dynamic Programming”. In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023.

[CGS23b] V. Coppé, X. Gillard, and P. Schaus. “Decision Diagram-Based Branch-and-Bound with Caching for Dominance and Suboptimality Detection”. In: (2023). arXiv: 2211.13118.

[CH13] A. A. Cire and W.-J. van Hoeve. “Multivalued decision diagrams for sequencing problems”. In: *Operations Research* 61.6 (2013), pp. 1411–1428.

[Cha+91] R. J. Chambers, R. L. Carraway, T. J. Lowe, and T. L. Morin. “Dominance and decomposition heuristics for single machine scheduling”. In: *Operations Research* 39.4 (1991), pp. 639–647.

[CS15] G. Chu and P. J. Stuckey. “Dominance breaking constraints”. In: *Constraints* 20 (2015), pp. 155–182.

[CS22] V. Coppé and P. Schaus. “A Conflict Avoidance Table for Continuous Conflict-Based Search”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 15. 1. 2022, pp. 264–266.

[DAF11] D. Datta, A. R. S. Amaral, and J. R. Figueira. “Single row facility layout problem using a permutation-based genetic algorithm”. In: *European Journal of Operational Research* 213.2 (2011), pp. 388–394.

[Dan57] G. B. Dantzig. “Discrete-Variable Extremum Problems”. In: *Operations Research* 5.2 (1957), pp. 266–277. (Visited on 10/25/2022).

[Dre87] Z. Drezner. “A heuristic procedure for the layout of a large number of facilities”. In: *Management Science* 33.7 (1987), pp. 907–915.

[Dum+95] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon. “An optimal algorithm for the traveling salesman problem with time windows”. In: *Operations research* 43.2 (1995), pp. 367–371.

[FR20] N. Frohner and G. R. Raidl. “Towards improving merging heuristics for binary decision diagrams”. In: *Learning and Intelligent Optimization: 13th International Conference, LION 13, Chania, Crete, Greece, May 27–31, 2019, Revised Selected Papers 13*. Springer. 2020, pp. 30–45.

[FS10] M. Fischetti and D. Salvagnin. “Pruning moves”. In: *INFORMS Journal on Computing* 22.1 (2010), pp. 108–119.

[Gil+21] X. Gillard, V. Coppé, P. Schaus, and A. A. Cire. “Improving the filtering of branch-and-bound MDD solver”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2021, pp. 231–247.

[Gil22] X. 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.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Books in mathematical series. W. H. Freeman, 1979.

[GL16] J. Guan and G. Lin. “Hybridizing variable neighborhood search with ant colony optimization for solving the single row facility layout problem”. In: *European Journal of Operational Research* 248.3 (2016), pp. 899–909.

[GMH20] R. Gentzel, L. Michel, and W.-J. van Hoeve. “HADDOCK: A language and architecture for decision diagram compilation”. In: *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26*. Springer. 2020, pp. 531–547.

[GMH22] R. Gentzel, L. Michel, and W.-J. van Hoeve. "Heuristics for MDD Propagation in HADDOCK". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022).* 2022.

[GMH23] R. Gentzel, L. Michel, and W.-J. van Hoeve. "Optimization Bounds from Decision Diagrams in Haddock". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research.* Springer. 2023, pp. 150–166.

[GS22] X. Gillard and P. Schaus. "Large Neighborhood Search with Decision Diagrams". In: *International Joint Conference on Artificial Intelligence.* 2022.

[GSC11] M. Garcia de la Banda, P. J. Stuckey, and G. Chu. "Solving talent scheduling with dynamic programming". In: *INFORMS Journal on Computing* 23.1 (2011), pp. 120–137.

[GSC21] X. Gillard, P. Schaus, and V. 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.* 2021, pp. 5243–5245.

[Gur22] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual.* 2022.

[Gus97] D. Gusfield. "Algorithms on stings, trees, and sequences: Computer science and computational biology". In: *Acm Sigact News* 28.4 (1997), pp. 41–60.

[HA92] S. S. Heragu and A. S. Alfa. "Experimental analysis of simulated annealing based algorithms for the layout problem". In: *European Journal of Operational Research* 57.2 (1992), pp. 190–202.

[Had+08] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. "Approximate compilation of constraints into multivalued decision diagrams". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2008, pp. 448–462.

[Hal70] K. M. Hall. "An r-dimensional quadratic placement algorithm". In: *Management science* 17.3 (1970), pp. 219–229.

[HDE14] H. He, H. Daume III, and J. M. Eisner. "Learning to search in branch and bound algorithms". In: *Advances in neural information processing systems* 27 (2014).

[Hel+14] M. Helmert, P. Haslum, J. Hoffmann, and R. Nissim. "Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces". In: *Journal of the ACM (JACM)* 61.3 (2014), pp. 1–63.

[HH06] T. Hadžić and J. N. Hooker. *Postoptimality analysis for integer programming using binary decision diagrams.* Tech. rep. Carnegie Mellon University, 2006.

[HH07] T. Hadžić and J. N. Hooker. "Cost-Bounded Binary Decision Diagrams for 0-1 Programming". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems.* Springer, 2007, pp. 84–98.

[HHH+07] M. Helmert, P. Haslum, J. Hoffmann, et al. "Flexible abstraction heuristics for optimal sequential planning". In: (2007).

[HHH10] S. Hoda, W.-J. van Hoeve, and J. N. Hooker. "A systematic approach to MDD-based constraint programming". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2010, pp. 266–280.

[HK62] M. Held and R. M. Karp. "A dynamic programming approach to sequencing problems". In: *Journal of the Society for Industrial and Applied mathematics* 10.1 (1962), pp. 196–210.

[HK88] S. S. Heragu and A. Kusiak. "Machine layout problem in flexible manufacturing systems". In: *Operations research* 36.2 (1988), pp. 258–268.

[HK91] S. S. Heragu and A. Kusiak. "Efficient models for the facility layout problem". In: *European Journal of Operational Research* 53.1 (1991), pp. 1–13.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[Hoe20] W.-J. van Hoeve. "Graph coloring lower bounds from decision diagrams". In: *International Conference on Integer Programming and Combinatorial Optimization.* Springer. 2020, pp. 405–418.

[Hoo13] J. N. Hooker. "Decision diagrams and dynamic programming". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research.* Springer. 2013, pp. 94–110.

[Hoo17] J. N. Hooker. "Job sequencing bounds from decision diagrams". In: *Principles and Practice of Constraint Programming: 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 23.* Springer. 2017, pp. 565–578.

[Hoo19] J. N. Hooker. “Improved Job Sequencing Bounds from Decision Diagrams”. In: *Principles and Practice of Constraint Programming*. Ed. by T. Schiex and S. de Givry. Vol. 11802. LNCS. Springer, 2019, pp. 268–283.

[Hor+21] M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg. “A*-based construction of decision diagrams for a prize-collecting scheduling problem”. In: *Computers & Operations Research* 126 (2021), p. 105125.

[HPS17] J. T. Haahr, D. Pisinger, and M. Sabbaghian. “A dynamic programming approach for optimizing train speed profiles with speed restrictions and passage points”. In: *Transportation Research Part B: Methodological* 99 (2017), pp. 167–182.

[HR13a] P. Hungerländer and F. Rendl. “A computational study and survey of methods for the single-row facility layout problem”. In: *Computational Optimization and Applications* 55.1 (2013), pp. 1–20.

[HR13b] P. Hungerländer and F. Rendl. “Semidefinite relaxations of ordering problems”. In: *Mathematical Programming* 140.1 (2013), pp. 77–97.

[HR21] M. Horn and G. R. Raidl. “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.* Springer. 2021, pp. 72–88.

[HS97] G. D. Hachtel and F. Somenzi. “A Symbolic Algorithms for Maximum Flow in 0-1 Networks”. In: *Formal Methods in System Design* 10.2 (1997), pp. 207–219.

[HT18] P. Hungerländer and C. Truden. “Efficient and easy-to-implement mixed-integer linear programs for the traveling salesperson problem with time windows”. In: *Transportation research procedia* 30 (2018), pp. 157–166.

[Hu95] A. J. Hu. “Techniques for efficient formal verification using binary decision diagrams”. PhD thesis. Stanford University, Department of Computer Science, 1995.

[Iba77] T. Ibaraki. “The power of dominance relations in branch-and-bound algorithms”. In: *Journal of the ACM (JACM)* 24.2 (1977), pp. 264–279.

[KB23] R. Kuroiwa and J. C. Beck. "Domain-independent dynamic programming: Generic state space search for combinatorial optimization". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 33. 1. 2023, pp. 236–244.

[KB90] T. Y.-k. Kam and R. K. Brayton. *Multi-valued decision diagrams*. Electronics Research Laboratory, College of Engineering, University of California, 1990.

[KD18] Z. Kalita and D. Datta. "A constrained single-row facility layout problem". In: *The international journal of advanced manufacturing technology* 98.5 (2018), pp. 2173–2184.

[KG14] R. Kothari and D. Ghosh. "An efficient genetic algorithm for single row facility layout". In: *Optimization Letters* 8.2 (2014), pp. 679–690.

[KH22] A. Karahalios and W.-J. van Hoeve. "Variable ordering for decision diagrams: A portfolio approach". In: *Constraints* 27.1 (2022), pp. 116–133.

[KH23] A. Karahalios and W.-J. van Hoeve. "Column Elimination for Capacitated Vehicle Routing Problems". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2023, pp. 35–51.

[KH67] R. M. Karp and M. Held. "Finite-state processes and dynamic programming". In: *SIAM Journal on Applied Mathematics* 15.3 (1967), pp. 693–718.

[KHL95] K. R. Kumar, G. C. Hadjinicola, and T.-l. Lin. "A heuristic procedure for the single-row facility layout problem". In: *European Journal of Operational Research* 87.1 (1995), pp. 65–73.

[KS74] W. H. Kohler and K. Steiglitz. "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems". In: *Journal of the ACM (JACM)* 21.1 (1974), pp. 140–156.

[Lan+93] A. Langevin, M. Desrochers, J. Desrosiers, S. Gélinas, and F. Soumis. "A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows". In: *Networks* 23.7 (1993), pp. 631–640.

[Lar+10] B. Larsen, E. Burns, W. Ruml, and R. Holte. "Searching without a heuristic: Efficient use of abstraction". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 24. 1. 2010, pp. 114–120.

[LBS15] A. Lieder, D. Briskorn, and R. Stolletz. “A dynamic programming approach for the aircraft landing problem with aircraft classes”. In: *European Journal of Operational Research* 243.1 (2015), pp. 61–69.

[LCL22] A. G. Labassi, D. Chételat, and A. Lodi. “Learning to compare nodes in branch and bound with graph neural networks”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 32000–32010.

[Lee59] C.-Y. Lee. “Representation of switching circuits by binary-decision programs”. In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[Liu+21] S. Liu, Z. Zhang, C. Guan, L. Zhu, M. Zhang, and P. Guo. “An improved fireworks algorithm for the constrained single-row facility layout problem”. In: *International Journal of Production Research* 59.8 (2021), pp. 2309–2327.

[LPV94] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. “EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.8 (1994), pp. 959–975.

[LW76] R. Love and J. Wong. “On solving a one-dimensional space allocation problem with integer programming”. In: *INFOR: Information Systems and Operational Research* 14.2 (1976), pp. 139–143.

[LZ23] J. H. Lee and A. Z. Zhong. “Exploiting functional constraints in automatic dominance breaking for constraint optimization”. In: *Journal of Artificial Intelligence Research* 78 (2023), pp. 1–35.

[MD15] C. Mears and M. G. De La Banda. “Towards automatic dominance breaking for constraint optimization problems”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence.* 2015.

[Min95] S.-i. Minato. *Binary decision diagrams and applications for VLSI CAD.* Vol. 342. Springer Science & Business Media, 1995.

[Mor+16] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. “Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning”. In: *Discrete Optimization* 19 (2016), pp. 79–102.

[MT23] K. Maier and V. Taferner. "Solving the constrained Single-Row Facility Layout Problem with Integer Linear Programming". In: *International Journal of Production Research* 61.6 (2023), pp. 1882–1897.

[Nik+21] B. Nikolic, A. Kartelj, M. Djukanovic, M. Grbic, C. Blum, and G. Raidl. "Solving the longest common subsequence problem concerning non-uniform distributions of letters in input strings". In: *Mathematics* 9.13 (2021), p. 1515.

[OH19] R. J. O'Neil and K. Hoffman. "Decision diagrams for solving traveling salesman problems with pickup and delivery in real time". In: *Operations Research Letters* 47.3 (2019), pp. 197–201.

[Pal17] G. Palubeckis. "Single row facility layout using multi-start simulated annealing". In: *Computers & Industrial Engineering* 103 (2017), pp. 1–16.

[PB96] J.-Y. Potvin and S. Bengio. "The vehicle routing problem with time windows part II: genetic search". In: *INFORMS Journal on Computing* 8.2 (1996), pp. 165–172.

[Pes+98] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau. "An exact constraint logic programming algorithm for the traveling salesman problem with time windows". In: *Transportation Science* 32.1 (1998), pp. 12–29.

[Pet03] J. Petit. "Experiments on the minimum linear arrangement problem". In: *Journal of Experimental Algorithmics* 8 (2003).

[Pis05] D. Pisinger. "Where are the hard knapsack problems?" In: *Computers & Operations Research* 32.9 (2005), pp. 2271–2284.

[PQ81] J.-C. Picard and M. Queyranne. "On the one-dimensional space allocation problem". In: *Operations Research* 29.2 (1981), pp. 371–391.

[PW06] Y. Pochet and L. A. Wolsey. *Production planning by mixed integer programming*. Vol. 149. 2. Springer, 2006.

[RCR18] M. Römer, A. A. Cire, and L.-M. Rousseau. "A local search framework for compiling relaxed decision diagrams". In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2018, pp. 512–520.

[RCR22] I. Rudich, Q. Cappart, and L.-M. Rousseau. “Peel-And-Bound: Generating Stronger Relaxed Bounds with Multivalued Decision Diagrams”. In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Ed. by C. Solnon. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[RCR23] I. Rudich, Q. Cappart, and L.-M. Rousseau. “Improved Peel-and-Bound: Methods for Generating Dual Bounds with Multivalued Decision Diagrams”. In: *Journal of Artificial Intelligence Research* 77 (2023).

[RS09] G. Righini and M. Salani. “Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming”. In: *Computers & Operations Research* 36.4 (2009), pp. 1191–1203.

[SCM21] K. Smith-Miles, J. Christiansen, and M. A. Muñoz. “Revisiting where are the hard knapsack problems? via instance space analysis”. In: *Computers & Operations Research* 128 (2021), p. 105184.

[SE10] H. Samarghandi and K. Eshghi. “An efficient tabu algorithm for the single row facility layout problem”. In: *European Journal of Operational Research* 205.1 (2010), pp. 98–105.

[SGW91] J. K. Suryanarayanan, B. L. Golden, and Q. Wang. “A new heuristic for the linear placement problem”. In: *Computers & Operations Research* 18.3 (1991), pp. 255–262.

[SH18] J. Seipp and M. Helmert. “Counterexample-guided Cartesian abstraction refinement for classical planning”. In: *Journal of Artificial Intelligence Research* 62 (2018), pp. 535–577.

[Sim69] D. M. Simmons. “One-dimensional space allocation: an ordering algorithm”. In: *Operations Research* 17.5 (1969), pp. 812–826.

[ST09] S. J. Shyu and C.-Y. Tsai. “Finding the longest common subsequence for multiple biological sequences by ant colony optimization”. In: *Computers & Operations Research* 36.1 (2009), pp. 73–91.

[SWH14] S. Sievers, M. Wehrle, and M. Helmert. “Generalized label reduction for merge-and-shrink heuristics”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 1. 2014.

[Weg00] I. Wegener. “Branching Programs and Binary Decision Diagrams: Theory and Applications”. In: *Discrete Applied Mathematics* (2000).