Discrete Optimization with Decision Diagrams: Design of a generic solver, Improved bounding techniques, and fast discovery of good feasible solutions with large neighborhood search

Xavier Gillard

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

October 2022

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Pierre **Schaus** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Peter **Van Roy**	UCLouvain/ICTEAM, Belgium
Pr. Charles **Pecheur**	UCLouvain/ICTEAM, Belgium
Pr. Daniele **Catanzaro**	UCLouvain/ICTEAM, Belgium
Pr. Willem-Jan **Van Hoeve**	Carnegie Mellon University, USA
Pr. André **Ciré**	University of Toronto, Canada

Discrete Optimization with Decision Diagrams: Design of a generic solver, Improved bounding techniques, and fast discovery of good feasible solutions with large neighborhood search

by Xavier Gillard

© Xavier Gillard 2022
ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

Abstract

Since its introduction by Bellmann in the 1950's Dynamic Programming (DP) has gained quite a lot of traction. Its simplicity and versatility have made it a tool of choice to solve all kinds of combinatorial problems. Over time, it has become so popular and successful that it is at the very heart of many classic algorithms taught to every computer science student around the world. For instance, dynamic programming is the base paradigm that underlies Dijkstra's shortest path algorithm. However, in spite of its inherent advantages, DP suffers from one major pitfall when solving NP-hard problems: in addition to requiring a potentially exponential amount of time in order to solve one such problem, a solver based on DP might require an exponential amount of memory as well. This in turn means that solving a hard combinatorial problem with DP might be intractable even for small to medium size problem instances.

In 2016 Bergman, Ciré, Van Hoeve and Hooker proposed a successful method combining *Dynamic Programming* with *Decision Diagrams* in a branch-and-bound framework. The strength of their approach stems from the ease of modeling which is leveraged from dynamic programming, and from the memory efficiency of the decision diagram data structure. The latter is a graphical model which provides an efficient encoding for exponentially sized sets of solutions to a given problem and allows the identification of an optimal solution in linear time.

This thesis proposes to deepen our knowledge of decision diagrams as a tool for optimization. It investigates their implementation and the trade-offs to make in order to benefit the most of their efficiency. It also investigates algorithmic ways to improve the performance of optimization solvers based on these graphical models. To that end, it proposes reasoning techniques to strengthen the bounds derived from each compiled decision diagram. Additionally, this thesis investigates ways to blend Decision Diagrams (DD) with other resolution techniques (e.g. Large Neighborhood Search, a.k.a LNS), and heuristics to swiftly find very good solutions to a given problem.

Acknowledgments

The popular wisdom tells that *it takes a village to raise a child.* As far as I am concerned, I really feel like completing Ph.D. proceeds from the same logic: it is – obviously – a scientific adventure, but it is just as much about human growth, learning about oneself, and shaping the person you are. This is why I would like to use these few lines to express my gratitude to the many people who have supported me throughout this incredible journey.

I would also like to address a huge thank you to Pierre, my advisor, who not only introduced me to discrete optimization, but also gave me the liberty to explore the topic as I liked, and at my pace. His guidance, ideas, and pieces of advice were essential to pull my research back on track when I felt like it was about to stall. His words were also kind and supportive when I faced troubled times, which truly meant a lot.

The experience would not have been the same if it weren't for my colleagues Charles, Vianney, Augustin, Guillaume, Alexandre, Hélène, Quentin, Christophe, Simon, Sophie, and Vanessa. It has been a pleasure to share that time with you, enjoying your company over coffee and lunch. Thank you all for making this Ph.D. such an enjoyable ride.

Additionally, I would also thank Guillaume and Benoit who have become more than just colleagues. I loved chatting, trolling, and exchanging ideas with them at the cafet' or on the train. Oftentimes, they have been the ones giving me the final push I needed to get things done.

I would also like to thank Charles who allowed me to pursue a Ph.D. Even if the two years we have spent working together have not been as fruitful as we wished, that period taught me a lot.

Nearing the end of these few lines, I would like to express all my gratitude to my parents without whom I would never have been able to complete this degree. Their dedication, care, and indefectible support have been essential to keeping me balanced and giving me the strength and means to go on with this project.

To conclude, I would like to thank Jennifer for her daily support, and my children Simon and Augustin for the spark of joy they bring to my life. I wish that my thesis could have somehow given you a taste for scientific matters...

Contents

Acknowledgments 3

Table of Contents 7

1 Introduction 1
- 1.1 Research Goals 2
- 1.2 Contributions 2
- 1.3 Publications 3
- 1.4 Outline 4

2 Background 5
- 2.1 Discrete optimization 5
- 2.2 Dynamic programming 5
- 2.3 Multivaluate Decision diagrams (MDD) 6
  - 2.3.1 Path 9
  - 2.3.2 Exact-MDD 10
  - 2.3.3 Bounded-Size Approximations 10
  - 2.3.4 Restricted-MDD: Under-approximation 11
    - 2.3.4.1 Bibliographic Note 11
  - 2.3.5 Relaxed-MDD: Over-approximation 12
    - 2.3.5.1 Cutset and Exact Custsets 13
  - 2.3.6 Summarizing Example 15
    - 2.3.6.1 Exact, Restricted and Relaxed MDD 16
    - 2.3.6.2 Exact Nodes and Cutsets 16
- 2.4 Branch-and-Bound with MDDs 16

3 Applications 19
- 3.1 The Maximum Independent Set Problem (MISP) 19
  - 3.1.1 DP Model 20
  - 3.1.2 Relaxation 21
  - 3.1.3 Correctness 21
- 3.2 Maximum Cut Problem (MCP) 22
  - 3.2.1 DP Model 22
  - 3.2.2 Relaxation 24
  - 3.2.3 Correctness 24

3.3 Maximum 2-Satisfiability Problem (MAX2SAT) 24
3.3.1 DP Model 25
3.3.1.1 Notations 26
3.3.1.2 Model 27
3.3.2 Relaxation 27
3.3.3 Correctness 28
3.4 Traveling Salesman Problem w. Time Windows (TSPTW) . . . 28
3.4.1 CP Model 28
3.4.2 DP Model 29
3.4.3 Relaxation 31
3.4.4 Bibliographic Note 32
3.5 Pigment Sequencing Problem (PSP) 33
3.5.1 DP Model 34
3.5.2 Relaxation 36
3.6 Description of the benchmark instances 36

4 How to Implement a Fast and Generic DDO solver ? 39

4.1 The *ddo* library 39
4.1.1 Why Rust ? 40
4.1.2 Parallel Computing 41
4.1.3 Discussion 43
4.2 Design of the *ddo* library 46
4.2.1 A Few Key Abstractions 48
4.2.2 A Complete Usage Example: Knapsack 52
4.3 Engineering the MDD data structure 55
4.3.1 The DecisionDiagram abstraction 55
4.3.2 Discussion on the Implementation Details of a DD . . 59
4.3.2.1 Deep MDD 59
4.3.2.2 Variant: Vector-based architecture 60
4.3.2.3 Pooled-MDD 63
4.3.2.4 Flat-MDD 65
4.3.2.5 Summarizing Example 67
4.4 Alternatives to *ddo* 69
4.5 Experimental Evaluation 71
4.5.1 Length of the r-t paths 72
4.5.2 Size of the Exact Cutset 73
4.5.3 Memory Usage 74
4.5.4 MDD Compilation Time 76
4.6 Conclusion 78

5 Improving the Filtering of Branch-and-Bound with MDD 81
5.1 Local bounds (LocB) . . . 81
5.1.1 Local Bound will not work with a shallow representation . . . 84
5.1.2 Complexity matters... . . . 86
5.2 Rough upper bound (RUB) . . . 86
5.2.1 Rough Upper Bounds for the Applications problems . 87
5.2.1.1 RUB for the MISP . . . 88
5.2.1.2 RUB for the MCP . . . 89
5.2.1.3 RUB for MAX2SAT . . . 92
5.2.1.4 RLB for the TSPTW . . . 95
5.2.1.5 RLB for the PSP . . . 97
5.3 Experimental Study . . . 98
5.4 Previous work . . . 102
5.5 Conclusion . . . 104

6 Large Neighborhood Search with Decision Diagrams 105
6.1 Motivation . . . 105
6.2 Large Neighborhood Search . . . 106
6.3 Experimental Study . . . 110
6.4 Related Work . . . 112
6.5 Conclusions . . . 113

7 A Global Minded Restricted DD Compilation Method 115
7.1 Motivation . . . 115
7.2 Intuition . . . 116
7.2.1 Visual Example . . . 118
7.3 Detailed Example . . . 118
7.4 Preprocessing: Compressing the instance . . . 121
7.4.1 Partitioning the original $\mathcal{I}$ in classes of similar items . 121
7.4.2 Rewriting a compressed instance . . . 121
7.5 During Resolution: Better Upper Bounds . . . 122
7.5.1 Phase 1: Compressing σ_t as σ'_t and solving it . . . 123
7.5.2 Phase 2: Inflating the optimal compressed solution . . 124
7.6 Evaluation . . . 124
7.7 Limitations . . . 126
7.8 Conclusions and further extensions . . . 127

8 Conclusion 129
8.1 Perspectives for Further Research . . . 130
8.1.1 Generalization of the Global Minded Restriction DD . 130
8.1.2 Investigate Alternate Merge Schemes . . . 131

8.1.3 Distributed parallelism 132
8.1.4 Integration with Other Kinds of Solvers 132

Introduction | 1

Whether we notice or not, operations research is ubiquitous in our everyday lives. Its use pervades varied domains such as mobile network coverage and redundancy insurance, production scheduling, organization of delivery tours and nurse rostering to name a few. The desire to solve hard problems is not novel and existed long before the advent of computers. For instance, in 1736 already, Euler investigated the famous *Königsberg Bridges* problem which is still taught today [Der16]. Later on, in 1857, Hamilton described the *Icosian Game* giving rise to the *Travelling Salesman Problem* which is still relevant and actively investigated today [ISO21]. The more specific desire of finding the *best* solution to such difficult problems is not new either. But the models, methods and algorithms which are used to process them are novel and in constant evolution.

Dynamic Programming, Constraint Programming, Mixed Integer Programming, SAT, and SMT and heuristic methods all contributed to improving the way we deal with these hard problems. Each of these methods brought its own set of strengths and weaknesses: be it in the form of ease of modeling, stronger reasoning, learning or by an increased flexibility in the way trade offs are arbitraged during the problem resolution.

Recently, Multivaluate Decision Diagrams (MDD) have drawn the attention of researchers in the constraint programming (CP) and operations research (OR) communities. These graphical models are a generalization of the *Binary Decision Diagrams* (BDD) which have long been used in the verification community e.g. for model checking purposes [Bur+92]. The popularity of these decision diagrams in the CP and OR communities stems from their ability to provide a compact representation of large solution spaces (e.g. in the case of the table constraint [PR15; VLS18]). One of the research streams which emerged from this increased interest about MDDs is *decision-diagrams based optimization* (DDO) [BC16b]. Its purpose is to efficiently solve combinatorial optimization problems by exploiting the structure of the problem being solved, which is achieved through the use of DDs.

A successful DDO paradigm was proposed by Bergman et al. who proposed to combine *Dynamic Programming* with *Decision Diagrams* in a branch-and-bound framework [Ber+16b]. The strength of their approach stems from the ease of modeling which is leveraged from dynamic programming, and

from the memory efficiency of the decision diagram data structure. The latter provides an efficient encoding for exponentially sized sets of solutions to a given problem, the best of which can be identified in linear time.

1.1 Research Goals

This thesis proposes to deepen our knowledge of decision diagrams as a tool for optimization. Their implementation and the trade offs to make in order to make the most of their efficiency. How to improve the quality of the solutions that are derived. How to blend them with other resolution techniques, deviating from the branch-and-bound framework which initially led to their being considered as powerful optimization tools.

1.2 Contributions

The main contributions of this thesis comprise:

- a generic and efficient framework for MDD-based optimization implemented in Rust. This framework offers both generic modeling facilities – staying close to the underlying mathematical model, heuristic definition and customization as well as parallel computation facilities.

- two bounding techniques to improve the filtering of branch-and-bound MDD solvers. One – *the rough upper bound* – which is used to speed up the compilation time tighten the bound derived from the compilation of a decision diagram; and one – *the local bound* – which exploits the structure of a compiled decision diagram to filter out nodes and hence helps avoiding thrashing during resolution. The implementation of these techniques is integrated to our generic framework. This implementation has been favorably compared to state-of-the-art discrete optimization solvers on various problems.

- a large neighborhood search procedure (and implementation) which adapts and leverages restricted decision diagrams to generate good quality neighborhoods around a feasible solution. This helps maintaining the strengths of MDD-based optimization and applying it where mere dynamic programming or branch-and-bound is not feasible.

- an alternate procedure, inspired by approximate dynamic programming to compile restricted MDDs which allows one to derive tighter, near-optimal bounds from a compiled decision diagram.

In addition to the above main contributions, this thesis also covers

- a new dynamic programming model to solve PSP.
- rough lower bounds for MISP, MCP, MAX2SAT, TSPTW, PSP.

1.3 Publications

Most of the work has already been published in:

- In Xavier Gillard, Pierre Schaus, and Vianney Coppé. *Ddo, a generic and efficient framework for MDD-based optimization.* IJCAI. 2020. we presented the generic framework which served as a vehicle to conduct the experiments related to the other contributions of this thesis. This was also the subject of a talk at the INFORMS 2021 annual meeting (Xavier Gillard. *"ddo" a Fast and Efficient Framework for Solving Combinatorial Optimization Problems with Branch-and-Bound MDD.* INFORMS Annual Meeting, Annaheim, CA – USA. Wednesday October 27, 2021.).
- In Xavier Gillard et al. "Improving the filtering of Branch-and-Bound MDD solvers". In: *CPAIOR*. 2021., we introduced the *rough lower bound* and *local bound* pruning techniques which we use to strengthen the capabilities of MDD-based solvers. This paper received the *Best Paper Award* at CPAIOR in July 2021 in Vienna – AT. In an extended version of the paper, we also introduced the rough upper bounds for the MISP, MCP, MAX2SAT and TSPTW.
- In Xavier Gillard and Pierre Schaus. *Large Neighborhood Search with Decision Diagrams.* IJCAI. 2022., we presented our contribution regarding the large neighborhood search method which builds upon decision diagrams. This is also the paper where we introduced a novel dynamic programming model for the pigment sequencing problem (PSP) and a rough lower bound procedure for that same problem.

In addition to the above, this doctoral research led to the following publications which are not directly related to *Discrete Optimization with Decision Diagrams.* This is why – and for the sake of presenting a coherent body of work, the presentation of these papers is omitted from current manuscript.

- In Xavier Gillard, Pierre Schaus, and Yves Deville. "Solvercheck: Declarative testing of constraints". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2019, pp. 565–582., we introduced SolverCheck: a property-based testing (PBT) library specifically designed to test CP solvers.
- In Xavier Gillard and Charles Pecheur. "On the community structure of SAT-BMC problems". In: *PhD Symposium at iFM'17 on Formal Methods:*

Algorithms, Tools and Applications (PhD-iFM'17). 2017., we investigated graph properties of SAT instances generated in the context of Bounded Model Checking. In particular, we investigated the community structure of these graphs and tried to identify whether these correspond to semantically meaningful information in the original higher level model.

1.4 Outline

The content of this dissertation is organized as follows: Chapter 2 gives a gentle introduction to the field of decision diagrams based optimization. It introduces dynamic programming, formalizes the decision diagrams and presents the branch-and-bound with MDD framework that has been proposed by Bergman et al. in 2016. Then, Chapter 3 continues with a presentation of the optimization problems that are used in the various experimental studies which have been used to validate the techniques proposed throughout this thesis. Subsequently, Chapter 4 investigates engineering concerns related to the implementation of a fast and generic library to build DD-based solvers. It presents several possible DD representations and discusses the performance impact of these choices. After that, Chapter 5 introduces reasoning techniques which help improve the filtering power of DD and hence the overall performance of DD based solvers. More specifically, it introduces a technique called *rough upper bound pruning* which uses a problem-specific bounding procedure. It also introduces *local bounds pruning* which is a technique exploiting the structure of the decision diagram to enhance the filtering power of the solver. The following chapter – Chapter 6, leaves the realm of exact optimization and proposes to hybridize DDO with meta heuristics in large neighborhood search framework. The latter provides an easy and convenient way of finding very good feasible solutions to an optimization problem quickly, at the expense of not always being able to prove the optimality of that solution. Eventually, Chapter 7 presents prospective work aiming to improve the resolution of a production planning problem through the use of a global minded approach to compiling decision diagrams. Finally, the last chapter summarizes the main message and results from this thesis and suggests some possible directions for future works.

Background

2

The coming sections give an overview of discrete optimization with decision diagrams as it was initially proposed by Bergman et al. in [Ber+16b]. These sections adopt a maximization perspective as is customary in the literature about optimization with decision diagrams. An orthogonal minimization perspective might equally have been adopted instead. The latter is achieved – without loss of generality – by flipping all signs.

2.1 Discrete optimization

A discrete optimization problem is a constraint *satisfaction* problem with an associated objective function to be maximized. The discrete optimization problem $\mathcal{P}$ is defined as $\max\{f(x) \mid x \in D \wedge C(x)\}$ where C is a set of constraints, $x = \langle x_0, \ldots, x_{n-1} \rangle$ is an assignment of values to variables, each of which has an associated finite domain D_i s.t. $D = D_0 \times \cdots \times D_{n-1}$ from where the values are drawn. In that setup, the function $f : D \rightarrow \mathbb{R}$ is the objective to be maximized. Among the set of feasible solutions $Sol(\mathcal{P}) \subseteq D$ (i.e. satisfying all constraints in C), we denote the optimal solution by x^*. That is, $x^* \in Sol(\mathcal{P})$ and $\forall x \in Sol(\mathcal{P}) : f(x^*) \geq f(x)$.

A subset of all discrete optimization problems exhibits an *optimal substructure*. That property states that the very nature of such problems imposes that an optimal solution to the global problem necessarily contains *within it* an optimal solution to subproblems[Cor+09, p. 327]. The problems having optimal substructure naturally lend themselves to resolution via a divide-and-conquer approach; and hence a dynamic programming formulation.

2.2 Dynamic programming

Dynamic Programming (DP) is a problem resolution strategy which has been introduced by Bellman in the mid 50's [Bel54]. It consists in a resolution of problems by decomposing them in smaller, simpler problems. This strategy is significantly popular and is at the heart of many classical algorithms (e.g., Dijkstra's algorithm [Cor+09, p.658] or Bellman-Ford's [Cor+09, p.651]).

Because of its divide-and-conquer nature, dynamic programming is often thought of in terms of recursion. However, it is also natural to consider it

as a labeled transition system. In that case, the *DP model* of a given discrete optimization problem $\mathcal{P}$ consists of:

- a set of state-spaces $S = \{S_0, \ldots, S_n\}$ among which one distinguishes the *initial state* r, the *terminal state* t and the *infeasible state* $\bot$.
- a set τ of transition functions s.t. $\tau_i : S_i \times D_i \rightarrow S_{i+1}$ for $i = 0, \ldots, n-1$ taking the system from one state s^i to the next state s^{i+1} based on the value d assigned to variable x_i (or to $\bot$ if assigning $x_i = d$ is infeasible). These functions should never allow one to recover from infeasibility ($\tau_i(\bot, d) = \bot$ for any $d \in D_i$).
- a set h of transition cost functions s.t. $h_i : S_i \times D_i \rightarrow \mathbb{R}$ representing the immediate reward of assigning some value $d \in D_i$ to the variable x_i for $i = 0, \ldots, n-1$.
- an initial value v_r.

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

$$
\begin{aligned}
&\text{maximize } f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i) \\
&\text{subject to} \\
&s^{i+1} = \tau_i(s^i, x_i) \text{ for } i = 0, \ldots, n-1; x_i \in D_i \wedge C(x_i) \\
&s^i \in S_i \text{ for } i = 0, \ldots, n
\end{aligned}
$$

where $C(x_i)$ is a predicate that evaluates to *true* when the partial assignment $\langle x_0, \ldots, x_i \rangle$ does not violate any constraint in C.

The appeal of such a formulation stems from its simplicity and its expressiveness which allows it to effectively capture the problem structure. Moreover, this formulation naturally lends itself to a multivaluate decision diagram (MDD) representation; in which case it represents an exact MDD encoding the complete set $Sol(\mathcal{P})$.

2.3 Multivaluate Decision diagrams (MDD)

At the heart of decision-diagrams based optimization (DDO), is the idea that DP transition systems lend themselves to materialization in the form of decision diagrams. In all generality, an MDD is a kind of layered automaton encoding sets of decision sequences. In that graph, a path between the source and a terminal node traverses one node from each layer [1] of the graph. In this

[1] Some authors introduce the possibility of adding so called long arcs which skip over one or several layers. For the sake of clarity, these will not be detailed here.

structure, the labels on the arcs connecting two nodes are interpreted as the assignment of a given value to a variable: the value being the label of the arc and the variable, the one associated to the layer crossed by the arc.

Because DDO aims at solving constraint *optimization* problems and not just constraint *satisfaction* problems, it uses a particular MDD flavor known as reduced weighted MDD – MDD as of now. As initially posed by Hooker [Hoo13], “MDDs can be perceived as a compact representation of the search trees. This is achieved, in this context, by superimposing isomorphic subtrees”.

Example 2.3.1 *A player wants to maximize the utility of his inventory in a video game. The inventory has a maximum capacity of 15kg, and the player has to select among the following items: a hammer, an axe, and a tent. The weight of these items is as follows 3kg for the hammer and the axe, and 12kg for the tent. Given the game is about survival, the player values the tent 120$, the hammer 15$, and the axe 12$. This problem is an easy-to-comprehend instance of a knapsack problem.*

This problem can be solved by repeatedly deciding to take one of the three items (decision variable) in the inventory or to leave it out. That is visually represented by the decision diagram from Figure 2.1. It is interesting to note that any path between the root and terminal nodes of that structure represents a feasible solution to the player's problem. Moreover, the DD not only encodes some *of the solutions, it encodes* all of them. *And as it happens, this representation is more compact than a search tree or a list encoding the same set of solutions. Indeed, the DD representation avoids the repetition of equivalent subproblems as is for instance the case of* $\{d, g, h, t\}$ *in Figure 2.1. In a table, these would correspond to repeated suffixes and in a search tree, it would provoke multiple repetitions of the same subtree without adding any new information.*

To define our MDD more formally, we will slightly adapt the notation from [BC16b]. A MDD $\mathcal{B}$ is a layered directed acyclic graph $\mathcal{B} = \langle n, U, A, l, v, \sigma \rangle$ where n is the number of variables from the encoded problem, U is a set of nodes, each of which is associated to some state $\sigma(u)$. The mapping $l : U \to \{0 \dots n\}$ partitions the nodes from U in disjoint layers $L_0 \dots L_n$ s.t. $L_i = \{u \in U : l(u) = i\}$ and the states of all the nodes belonging to the same layer pertain to the same DP-state-space ($\forall u \in L_i : \sigma(u) \in S_i$ for $i = 0, \dots, n$). Also, it should be the case that a layer only ever contains one node per class of equivalent states ($\forall u_1, u_2 \in L_i : u_1 \neq u_2 \implies \sigma(u_1) \nsim \sigma(u_2)$, for $i = 0, \dots, n$). The enforcement of this property, however, cannot be done efficiently. Indeed, equivalence checking is known to be NP-hard [CGP99][2].

[2]Two states $\sigma(u_1)$ and $\sigma(u_2)$ are equivalent if the MDD they define encode the exact same set of partial solutions. In other words, $\sigma(u_1) \sim \sigma(u_2)$ iff the set of u_1-t

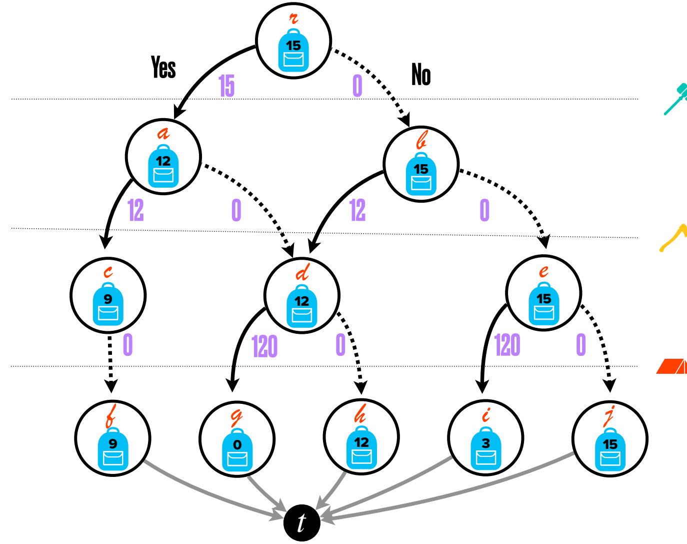


Figure 2.1: Visual Example For a Knapsack Problem with three decision variables: a hammer with weight 3 and value 15, an axe with weight 3 and value 12, and a tent with weight 12 and value 120 (Example 2.3.1).

That is, enforcing the single-node-per-equivalence-class property is as hard a problem as the resolution of a constraint optimization problem. This is why this thesis uses a slightly relaxed version of that property. The latter only forbids the co-occurrence of two nodes with equal states inside of one same layer rather than the co-occurrence of two nodes with equivalent states ($\forall u_1, u_2 \in L_i : u_1 \neq u_2 \implies \sigma(u_1) \neq \sigma(u_2)$, for $i = 0, \ldots, n$).

The set $A \subseteq U \times U \times D$ from our formal model is a set of labeled directed arcs connecting the nodes from U. Each such arc $a = (u_1, u_2, d)$ connects nodes from subsequent layers ($l(u_1) = l(u_2) - 1$) and should be regarded as the materialization of a branching decision d about variable $x_{l(u_1)}$. Thus, the decision d is a value drawn from the domain of the variable $x_{l(u_1)}$. In addition to the decision value d, the arcs from this formal model are also annotated with a weight information. The latter, however, does not participate in the *identity* of the arc in question; it is fully determined by the mapping $v : A \rightarrow \mathbb{R}$. In order to ease the reading of this thesis, we will use the notation $d(a)$ to denote the decision component of an arc a and its weight as $v(a)$.

Example 2.3.2 *In Figure 2.1, the set U corresponds to the set $\{r, a, b, c, d, e, f,$*

---

paths is equal to the set of u_2-t paths. That is, $\sigma(u_1) \sim \sigma(u_2)$ if for any sequence of arcs $((u_1, a_1, d_1), (a_1, a_2, d_2), \ldots, (a_k, t, d_k))$ with weights $(v_1, v_2, \ldots, v_k)$, there exists a corresponding sequence of arcs $\left((u_2, a'_1, d_1), (a'_1, a'_2, d_2), \ldots, \left(a'_k, t, d_k\right)\right)$ with the same weights $(v_1, v_2, \ldots, v_k)$ and vice-versa.

$g, h, i, j, t\}$. The state $\sigma(c)$ associated with node c is a sack capable of carrying an additional 9 kilograms of weight, and that of node i = $\sigma(i)$ is a sack capable of bearing 3 kg extra.

The arc between nodes r and a is the arc (r, a, yes) and is must be understood as the decision $[\![hammer = yes]\!]$ performed from a state $\sigma(r)$ where the sack is empty (its capacity is still 15kg). It should also be understood that the weight of that arc (= the benefit of taking the hammer) is 15$.

2.3.1 Path

A path p of some MDD $\mathcal{B}$ is a sequence $a_i \dots a_k$ of arcs in $\mathcal{B}$ s.t the originating node of one arc is the terminal node of the previous one. In order to express this definition formally, we pose that for any arc $a = (u_1, u_2, d)$, the expression $o(a) = u_1$ denotes the originating end of the arc and $e(a) = u_2$ denotes its terminal end. On that basis, a path $p = a_i \dots a_k$ is a sequence of arc s.t. $o(a_j) = e(a_{j-1}), \forall i < j \leq k$.

Given that, in a MDD, an arc a is interpreted as a decision made about a problem variable; any path can be interpreted as a (partial) assignment on these variables. It is thus the case that all paths $p = a_i \dots a_k$ have a corresponding assignment $x^p = \{[\![x_j = d(a_j)]\!] \mid i < j \leq k\}$. By analogy to the v relationship which specifies the *weight* of an arc, the weight of path p is denoted by $v(p) = \sum_{j=i}^{k} v(a_j)$.

Any path $p = a_0 \dots a_n$ is said to be an *r-t* path iff $o(a_0) = r$ and $e(a_n) = t$. That is, an *r-t* path is a path traversing the complete MDD: starting at the root of the diagram and finishing at a terminal node. By extension of the relationship $v(\cdot)$ defined on arcs and paths, $v^*(\mathcal{B}) = \max\{v(p) \mid p \in \mathcal{B}\}$ is used to denote the weight of a longest r-t path in $\mathcal{B}$. Since each *r-t* path p describes an assignment that satisfies $\mathcal{P}$, we have $x^p \in Sol(\mathcal{P})$. Because of this correspondance, we will use $Sol(\mathcal{B})$ to denote the set of all the solutions encoded in the r-t paths of MDD $\mathcal{B}$. Also, because unsatisfiability is irrecoverable, r-$\perp$ paths are typically omitted from MDDs. It follows that a nice property from using a MDD representation $\mathcal{B}$ for the DP formulation of a problem $\mathcal{P}$, is that finding x^* is as simple as finding the longest r-t path p in $\mathcal{B}$ according to its weight $v(p)$.

Example 2.3.3 *The longest r-t path in Figure 2.1 is (r, a, d, g, t) and its weight is 135. That path encodes the optimal solution to the player's problem from Example 2.3.1. Namely, it corresponds to the solution $[\![hammer = yes, axe = no, tent = yes]\!]$.*

2.3.2 Exact-MDD

For a given problem $\mathcal{P}$, an exact MDD $\mathcal{B}$ is an MDD that exactly encodes the solution set $Sol(\mathcal{B}) = Sol(\mathcal{P})$ of the problem $\mathcal{P}$. In other words, not only do all r-t paths encode valid solutions of $\mathcal{P}$, but no feasible solution is present in $Sol(\mathcal{P})$ and not in $\mathcal{B}$. Also, the weights of the arcs in $\mathcal{B}$ exactly correspond to the value of the transition cost functions of $\mathcal{P}$. It follows that $v(p) = f(x^p) - v_r$ for all paths p in $\mathcal{B}$. In particular, the weight of a longest r-t path p^* immediately yields the optimal value of $\mathcal{P}$ ($v^*(\mathcal{B}) = v(p^*) = f(x^*) - v_r$).

An exact MDD for $\mathcal{P}$ can be compiled in a top-down fashion[3]. This naturally follows from the above definition. To that end, one simply proceeds by a repeated unrolling of the transition relations until all variables are assigned; as shown in Algorithm 1.

Algorithm 1 Top Down Compilation of an Exact MDD

```
1:  Input: a DP-model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  L_0 ← {r}
3:  for i ∈ {0 ... n − 1} do
4:      for u ∈ L_i, d ∈ D_i do
5:          u′ ← a node associated with state τ_i(σ(u), d)
6:          if σ(u′) ≠ ⊥ then
7:              U ← U ∪ {u′}
8:              L_{i+1} ← L_{i+1} ∪ {u′}
9:              a ← (u, u′, d)
10:             v(a) ← h_i(σ(u), d)
11:             A ← A ∪ {a}
```

2.3.3 Bounded-Size Approximations

In spite of the compactness of their encoding, the construction of MDD suffers from a potentially exponential memory requirement in the worst case[4]. Thus, using MDDs to exactly encode the solution space of a problem is often intractable. Therefore, one must resort to the use of *bounded-size* approximation of the exact MDD. These are compiled generically by inserting a call to a width-bounding procedure to ensure that the width (the number $|L_i|$ of distinct nodes belonging to the layer L_i) of the current layer L_i does not exceed a given bound W. Depending on the behavior of that procedure, one can either

[3] An incremental refinement *a.k.a. construction by separation* procedure is detailed in [Cir14, pp. 51–52]

[4] Consequently, it also suffers from a potentially exponential time requirement in the worst case. Indeed, time is constant in the final number of nodes (unless the transition functions themselves are exponential in the input).

compile a restricted-MDD (= an under-approximation) or a relaxed-MDD (= an over-approximation).

2.3.4 Restricted-MDD: Under-approximation

Restricted DD were first introduced by Bergman et al. in [Ber+14a]. Such a decision diagram provides a bounded-width under-approximation of some exact-MDD. As such, all paths of a restricted-MDD encode valid solutions, but some solutions might be missing from the MDD. This is formally expressed as follows: given the DP formulation of a problem $\mathcal{P}$, $\mathcal{B}$ is a restricted-MDD iff $Sol(\mathcal{B}) \subseteq Sol(\mathcal{P})$. It must thus be the case that $v(p) = f(x^p) - v_r$ for all paths $p \in \mathcal{B}$, as if $\mathcal{B}$ were exact. However, because the paths of $\mathcal{B}$ might not cover the complete solution set of $\mathcal{P}$, it is possible that no longest path p^* of $\mathcal{B}$ correspond to the optimal assignment x^*. Hence, the value $v^*(\mathcal{B})$ yields a *lower bound* on the optimal value of $\mathcal{P}$ ($v^*(\mathcal{B}) \leq f(x^*) - v_r$).

As shown in Algorithm 2, restricted-MDD are compiled in a top down fashion similar to the compilation of exact-MDDs (faded). The only difference with the compilation of an exact-MDD being the potential call to a *restrict* procedure guarded by the check $|L_{i+1}| > W$ to ensure that no layer of the resulting MDD be larger than the imposed limit W (highlighted).

The behavior of the restrict procedure is not imposed. It suffices for its correctness that it deletes nodes from the current layer until its width fits within the specified bound W. In practice, the procedure simply selects and deletes a subset of the nodes from L_{i+1} which are heuristically assumed to have the smallest impact on the tightness of the bound derived from the MDD. Various heuristics have been studied in the literature [Ber+14b], and the heuristic that decides to select (hence remove) the nodes having the shortest longest path from the root (*minLP*) was shown to be the best performing heuristic in practice.

2.3.4.1 Bibliographic Note

It is worth mentioning that the compilation of a restricted-MDD is equivalent to an application of the restricted dynamic programming approach which has been proposed by Baldacci et al. in [MD96] in the context of the time-dependent traveling salesman problem (TD-TSP).

2.3.5 Relaxed-MDD: Over-approximation

Relaxed DDs were first introduced by [And+07] as an alternative representation of the domain store used in constraint programming. It was only a few years later that [BHH11; Ber+14b] proposed to use these structures as a means to derive bounds on optimization problems. In line with these recent

Algorithm 2 Top Down Compilation of a Restricted DD

1: **Input:** a DP-model $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$
2: **Input:** a maximum layer width W
3: $L_0 \leftarrow \{r\}$
4: **for** $i \in \{0 \ldots n-1\}$ **do**
5:     **for** $u \in L_i, d \in D_i$ **do**
6:         $u' \leftarrow$ a node associated with state $\tau_i(\sigma(u), d)$
7:         **if** $\sigma(u') \neq \bot$ **then**
8:             $U \leftarrow U \cup \{u'\}$
9:             $L_{i+1} \leftarrow L_{i+1} \cup \{u'\}$
10:             $a \leftarrow (u, u', d)$
11:             $v(a) \leftarrow h_i(\sigma(u), d)$
12:             $A \leftarrow A \cup \{a\}$
13:     **if** $|L_{i+1}| > W$ **then**
14:         $L_{i+1} \leftarrow restrict(L_{i+1})$

pieces of work, we will consider that relaxed-MDD $\mathcal{B}$ provides a bounded-width over-approximation of some exact-MDD. As such, it may hold paths that are no solution to $\mathcal{P}$, the problem being solved. We have thus formally that $Sol(\mathcal{B}) \supseteq Sol(\mathcal{P})$. Also, it must be the case that $v(p) \geq f(x^p) - v_r$ for any path $p \in \mathcal{B}$. It follows that the solution represented by a longest r-t path p^* of $\mathcal{B}$ is not guaranteed to be x^*. The assignment x^{p^*} is not even guaranteed to be a feasible solution according to the constraints of the problem ($C(x^{p^*})$ might evaluate false). Nevertheless, the length of p^* is guaranteed to yield an upper bound on the optimal value ($v^*(\mathcal{B}) \geq f(x^*) - v_r$).

Algorithm 3 details a top down procedure to compile relaxed-MDDs. This procedure is fairly similar to the ones used to compile exact and restricted MDDs (faded lines are common to all three algorithms). The major difference stems from the call to the *relax* width-bounding routine. The purpose of *relax* is to replace a subset of the nodes of a layer whose width exceeds the limit W by a new node standing for all of them. This is why compiling a relaxed-MDD requires one to be able to *merge* several nodes into an inexact one. To that end, two operators are used:

- $\oplus$ which yields a new node combining the states of a selection of nodes so as to over-approximate the states reachable in the selection.
- Γ which is used to possibly relax the weight of arcs incident to the selected nodes.

These operators are used as shown in Algorithm 4: the width-bounding procedure starts by heuristically selecting the least promising nodes. Then the

Algorithm 3 Top Down Compilation of a Relaxed DD

1: **Input:** a DP-model $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$
2: **Input:** a maximum layer width W
3: **Input:** a node merging operator $\oplus$
4: **Input:** an arc relaxation operator Γ
5: $L_0 \leftarrow \{r\}$
6: **for** $i \in \{0 \ldots n-1\}$ **do**
7:     **for** $u \in L_i, d \in D_i$ **do**
8:         $u' \leftarrow$ a node associated with state $\tau_i(\sigma(u), d)$
9:         **if** $\sigma(u') \neq \bot$ **then**
10:             $U \leftarrow U \cup \{u'\}$
11:             $L_{i+1} \leftarrow L_{i+1} \cup \{u'\}$
12:             $a \leftarrow (u, u', d)$
13:             $v(a) \leftarrow h_i(\sigma(u), d)$
14:             $A \leftarrow A \cup \{a\}$
15:     **if** $|L_{i+1}| > W$ **then**
16:         $L_{i+1} \leftarrow relax(L_{i+1}, U, A, \oplus, \Gamma)$

states of these selected nodes are combined with one another so as to create a merged node $\mathcal{M} = \oplus(selection)$. After that, the inbound arcs incident to all selected nodes are Γ-relaxed and redirected towards $\mathcal{M}$. Finally, the result of the merger ($\mathcal{M}$) is added to the layer L in replacement of the initial selection of nodes.

Algorithm 4 Relax Procedure

1: **Input:** L the layer that needs to be relaxed
2: **Input:** the set U of nodes existing in the MDD being compiled
3: **Input:** the set A of arcs existing in the MDD being compiled
4: **Input:** $\oplus$ a node merging operator
5: **Input:** Γ an arc relaxation operator
6: $selection \leftarrow HeuristicSelection(L)$
7: $\mathcal{M} \leftarrow \oplus(selection)$
8: **for all** $a \in A; e(a) \in selection$ **do**
9:     $v(a) \leftarrow \Gamma(a, \mathcal{M})$
10:     $e(a) \leftarrow \mathcal{M}$
11: $U \leftarrow U \setminus selection \cup \{\mathcal{M}\}$
12: $L \leftarrow L \setminus selection \cup \{\mathcal{M}\}$

2.3.5.1 Cutset and Exact Custsets

Considering that Restricted DD and Relaxed DD provide bounded-width *approximations* of what would be the Exact DD of the (sub-) problem at their root, the devising of a *complete* algorithm capable of yielding the globally-optimal-solution requires that one be capable of exploring the portions of the state-spaces that have not been exactly covered by the approximations. A naive approach to satisfying this requirement would simply consist in the repeated development of all direct-children of the approximate DD's root node. A better approach is possible however. In [Ber+16b], Bergman et al. proposed to instead exploit the information from the approximate DD and enumerate the residual subproblems defined by the nodes forming a border separating the approximate DD and its exact counterpart. In mathematical parlance, this set of nodes forming a frontier up to which an exact and approximate DD have not diverged, is called an *exact cutset.*

Cutset More formally, a *cutset* for some MDD $\mathcal{B}$ is a subset C of the nodes from $\mathcal{B}$ such that any $r-t$ path of $\mathcal{B}$ goes through at least one node $\in C$. Put another way, a cutset C of some MDD $\mathcal{B}$ is a subset of the nodes of $\mathcal{B}$ such that the source node (r) and terminal nodes (t) of $\mathcal{B}$ are disconnected when C is removed. Obviously there may exist many different cutsets for the same MDD.

Exact Node In a relaxed MDD, a node u is said to be *exact* iff all its incoming paths lead to the same state $\sigma(u)$. That is, u is exact iff neither itself nor any of its ancestors is the result of a merge operation.

Exact Cutset From there, an exact cutset of some relaxed MDD is simply a cutset whose nodes are all exact. Based on this definition, it is easy to convince oneself that an exact cutset constitutes a frontier up to which a relaxed MDD and its exact counterpart have not diverged.

Any relaxed-MDD admits at least one exact cutset – e.g. the trivial $\{r\}$ case. Often though, it is not unique and different types of exact cutsets have been studied in the literature. For instance:

- *First Exact Layer* (FEL) cutset. This is the naive approach mentioned in the introduction. It mimics the traditional branching scheme by using the *shallowest* possible exact cutset. Formally, given a relaxed MDD $\mathcal{B}$,

$$FEL(\mathcal{B}) = L_1$$

- *Last Exact Layer* (LEL) cutset which selects as exact cutset the *deepest*

layer comprising only exact nodes. Formally, given a relaxed MDD $\mathcal{B}$:

$$LEL(\mathcal{B}) = L_i : \forall u \in L_i : u \text{ is exact} \wedge \nexists L_j : j > i \text{ s.t. } \forall u' \in L_j : u' \text{ is exact}$$

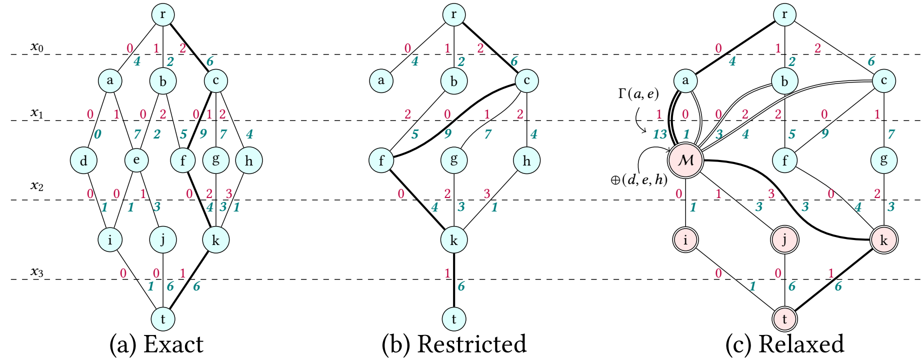
- *Frontier Cutset* (FC) which selects as exact cutset all the exact nodes of the relaxed MDD which are the direct parent of an inexact node. Formally, given a relaxed MDD $\mathcal{B}$,

$$FC(\mathcal{B}) = \{u \mid u \in U : u \text{ is exact} \wedge \exists (u, u') \in A : u' \text{ is not exact}\}.$$

Bergman et al. have experimentally shown in [Ber+16b] that most of the time, LEL is superior to all other exact cutsets in practice.

2.3.6 Summarizing Example

Figure 2.2 summarizes the information from section 2.3. It displays the three MDDs corresponding to one same example problem having four variables.



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

Figure 2.2: The exact (a), restricted (b) and relaxed (c) versions of an MDD with four variables. The width of MDDs (b) and (c) have been bounded to a maximum layer width of three. The decision labels of the arcs are shown above the layers separation lines (dashed). The arc weights are shown below the layer separation lines. The longest path of each MDD is boldfaced. In (c), the node $\mathcal{M}$ is the result of merging nodes d, e and h with the $\oplus$ operator. Arcs that have been relaxed with the Γ operator are pictured with a double stroke. Note, because these arcs have been Γ-relaxed, their value might be greater than that of corresponding arcs in (a), (b). Similarly, all "inexact" nodes feature a double border.

2.3.6.1 Exact, Restricted and Relaxed MDD

The exact MDD (a) encodes the complete solution set and, equivalently, the state space of the underlying DP encoding. One easily notices that the restricted DD (b) is an under approximation of (a) since it achieves its width

boundedness by removing nodes d and e and their children (i, j). Among others, it follows that the solution $[x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0]$ is not represented in (b) even though it exists in (a). Conversely, the relaxed diagram (c) achieves a maximum layer with of 3 by merging nodes d, e and h into a new inexact node $\mathcal{M}$ and by relaxing all arcs entering one of the merged nodes. Because of this, (c) introduces solutions that do not exist in (a) as is for instance the case of the assignment $[x_0 = 0, x_1 = 0, x_2 = 3, x_3 = 1]$. Moreover, because the operators $\oplus$ and Γ are correct[5], the length of the longest path in (c) is an upper bound on the optimal value of the objective function. Indeed, one can see that the length of the longest path in (a) (= the exact optimal solution) has a value of 25 while it amounts to 26 in (c).

2.3.6.2 Exact Nodes and Cutsets

In Figure 2.2 (c), the first inexact node $\mathcal{M}$ occurs in layer L_2. Hence, the LEL cutset comprises all nodes (a, b, c) from the layer L_1. An FC cutset for (c) comprises the nodes (a, b, c, f, g) since each of these nodes is the immediate parent of an inexact node. Indeed, $\mathcal{M}$ was created by merging nodes (d, e, h) from the exact MDD which makes it an inexact node. And, because $\mathcal{M}$ is a parent of nodes i, j and k, these three nodes are considered inexact too.

2.4 Branch-and-Bound with MDDs

As explained in sections 2.3.4 and 2.3.5, restricted and relaxed MDDs provide a convenient way of deriving lower and upper bounds on the optimal value of the optimization problem $\mathcal{P}$ they represent. While being able to derive good lower and upper bounds for $\mathcal{P}$ is useful when the goal is to use these bounds to strengthen algorithms [DH18; Tja18; TH19]; it is not the only way these approximations can be used. A complete and efficient branch-and-bound algorithm relying on those approximations was proposed in [Ber+16b] which is hereby reproduced (Algorithm 5).

This algorithm works as follows: at start, the node r is created for the initial state of the problem and placed onto the *fringe* – a global priority queue that tracks all nodes remaining to explore and orders them from the most to least promising. Then, a loop consumes the nodes from that fringe (line 4), one at a time and explores it until the complete state space has been exhausted. The *exploration* of a node u inside that loop proceeds as follows: first, one compiles a restricted DD $\underline{\mathcal{B}}$ for the sub-problem rooted in u (line 8). Because all paths in a restricted DD are feasible solutions, when the lower bound $v^*(\underline{\mathcal{B}})$ derived from the restricted DD $\underline{\mathcal{B}}$ improves over the current best

[5] The very definition of these operators is problem-specific. However, [Hoo17] formally defines the conditions that are necessary to correctness.

known solution $\underline{v}$; then the longest path of $\underline{\mathcal{B}}$ (best sol. found in $\underline{\mathcal{B}}$) and its length $v^*(\underline{\mathcal{B}})$ are memorized (lines 10-12).

In the event where $\underline{\mathcal{B}}$ is exact (no restriction occurred during the compilation of $\underline{\mathcal{B}}$), it covers the complete state space of the sub-problem rooted in u. Which means the processing of u is complete and we may safely move to the next node. When this condition is not met, however, some additional effort is required. In that case, a *relaxed* DD $\overline{\mathcal{B}}$ is compiled from u (line 14). That relaxed DD serves two purposes: first, it is used to derive an upper bound $v^*(\overline{\mathcal{B}})$ which is compared to the current best known solution (line 15). This gives us a chance to prune the unexplored state space under u when $v^*(\overline{\mathcal{B}})$ guarantees it does not contain any better solution than the current best. The second use of $\overline{\mathcal{B}}$ happens when $v^*(\overline{\mathcal{B}})$ cannot provide such a guarantee. In that case, the exact cutset of $\overline{\mathcal{B}}$ is used to enumerate residual sub-problems which are enqueued onto the fringe (lines 16-17). And because – by definition, an exact cutset of $\overline{\mathcal{B}}$ is a cutset of $\mathcal{B}$ as well, the nodes it contains cover all paths from both $\mathcal{B}$ and $\overline{\mathcal{B}}$. This guarantees that portions of the state space of $\mathcal{B}$ which have not been explored in $\overline{\mathcal{B}}$ can be retrieved from the exact cutset nodes. This in turn guarantees the completeness of Algorithm 5 [Ber+16b].

Algorithm 5 Branch-And-Bound with DD

```
1:  Input: a DP-model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  Input: a node merging operator ⊕
3:  Input: an arc relaxation operator Γ
4:  Create node r and add it to Fringe
5:  x_ ← ⊥
6:  v_ ← −∞
7:  while Fringe is not empty do
8:      u ← Fringe.pop()
9:      B_ ← Restricted(u)
10:     if v*(B_) > v_ then
11:         v_ ← v*(B_)
12:         x_ ← x*(B_)
13:     if B_ is not exact then
14:         B¯ ← Relaxed(u, ⊕, Γ)
15:         if v*(B¯) > v_ then
16:             for all u′ ∈ B¯.exact_cutset() do
17:                 Fringe.add(u′)
18: return (x_, v_)
```

Applications | 3

Contributions and Publication Information

The models presented in this chapter have not been the subject of a dedicated publication. Indeed, the models for MISP, MCP and MAX2SAT are the ones proposed by Bergman et al. in [Ber+16b] and [Ber+16a]. These should thus be considered as background information related to the problems used when validating the techniques presented in subsequent chapters. This also apply to the PSP model from section 3.5 – to some extent. The DP formulation for that problem, however, is novel and has been introduced in Xavier Gillard and Pierre Schaus. *Large Neighborhood Search with Decision Diagrams.* IJCAI. 2022.

This chapter describes the problems that are used to evaluate the techniques proposed throughout this manuscript. In practice, it covers the following five problems:

1. the Maximum Independent Set Problem (MISP),
2. the Maximum Cut Problem (MCP),
3. the Maximum 2 Satisfiability Problem (MAX2SAT),
4. the Travelling Salesman Problem with Time Windows (TSPTW)
5. the Pigment Sequencing Problem (PSP)

For each problem, it describes the problem, its DP formulation as well as the merge ($\oplus$) and relaxation (Γ) operators involved in the compilation of relaxed DD as shown per Algorithm 3 in Chapter 2. After that, a description of the benchmark instances that are used in the experimental validations of all techniques proposed in this manuscript is given.

3.1 The Maximum Independent Set Problem (MISP)

In graph theory, the problem of finding an Independent Set consists in finding a subset of vertices in a graph such that no edge exists in the graph that con-

nects two of the selected nodes. By extension, the *Maximum* (Weighted) Independent Set Problem, consists of finding an Independent Set in a weighted graph such that the sum of the weight of the selected nodes is maximal. This problem is famous – among others – because it is equivalent to finding a clique of maximum weight in the complement of the instance graph.

In spite of its apparently purely theoretical nature, the MISP has been used in many real world applications. For instance, in bioinformatics [Ebl+12], data mining [ESB99] and the analysis of social networks[BBH11].

Formally, given a weighted graph $G = (V, E, w)$ where $V = \{0, 1, \ldots, n-1\}$ is a set of vertices, $E \subseteq V \times V$ the set of edges connecting those vertices and $w = \{w_0, w_1, \ldots, w_{n-1}\}$ is a set of weights s.t. w_i is the weight of node i; the MISP can be expressed as follows:

$$\max \sum_{i=0}^{n-1} w_i x_i \tag{3.1}$$

$$x_i \in \{0, 1\}\, \forall i \in V \tag{3.2}$$

$$x_i + x_j \leq 1 \forall (i, j) \in E \tag{3.3}$$

In this formulation, a boolean variable x_i is used to tell whether or not the vertex i is selected in the solution independent set (3.2). The summation (3.1) denotes the objective function to maximize and (3.3) is a constraint that enforces the absence of edge between any two selected vertices.

3.1.1 DP Model

In [Ber+16a, pp.33-34], Bergman et al. propose a MISP DP model in which a state s^i of the i-th layer is a set representing the vertices that might possibly participate in the maximum independent set. Therefore, the initial state comprises all vertices since any vertex can – a priori – belong to the solution. Similarly, the terminal node t is the empty set as no vertex can be added to the solution after all decisions have been made.

The transition relation proceed from the same logic: when deciding about the inclusion of vertex i in the independent set, two cases are possible given a state s^i. If $i \in s^i$, one can decide to include i in the solution or to leave it out. Otherwise i cannot be included in the independent set. Should the decision be made to include i in the solution, then vertex i and all its neighbors ($N(i) = \{j \in V \mid (i, j) \in A\}$) must be removed from the candidates list. The successor state of s^i is thus $s^{i+1} = s^i \setminus (\{i\} \cup N(i))$. If on the other hand, the decision is made to not include i in the solution; it simply means that i is the only vertex that needs to be removed from the candidates list. Hence, the successor state of s^i becomes $s^{i+1} = s^i \setminus \{i\}$.

The transition cost is straightforward in this model as it reflects the intuition: a decision to add the vertex i to the independent set increases the objective function by w_i whereas a decision to not include it leaves the objective unchanged.

This model is formally expressed as follows:

- State spaces: $S_k = \left\{s^k \in 2^V \mid j \notin s^k, 0 \leq j < k\right\}$ with the initial state $r = V$ and a terminal state $t = \emptyset$
- State transition: $\tau_k(s^k, d) = \begin{cases} s^k \setminus (N(k) \cup \{k\}) & \text{when } d = 1 \\ s^k \setminus \{k\} & \text{when } d = 0 \end{cases}$
- Transition cost: $h_k(s^k, d) = \begin{cases} w_k & \text{when } d = 1 \\ 0 & \text{when } d = 0 \end{cases}$
- Root value $v_r = 0$

3.1.2 Relaxation

As explained in Chapter 2, the compilation of *relaxed* DDs, requires that two relaxation operators be defined: a merge operator $\oplus$ and an arc relaxation operator Γ. These must respectively guarantee that a) no feasible solution is removed from the relaxed DD consecutively to a merge operation and b) the length of an r-t path passing through a merged node of a relaxed DD be at least as long as all the paths it stands for in the exact counterpart of that DD. In practice, in the context of the MISP DP model presented above, it means that when merging several nodes, all the vertices that are considered as potential members of the maximum independent set by any of the nodes being merged must belong to the resulting merged state ($\oplus(\mathcal{M}) = \bigcup_{u \in \mathcal{M}} u$). The arc relaxation operator leaves the weight of an arc unchanged. Formally, we have thus:

- Merge Operator: $\oplus(\mathcal{M}) = \bigcup_{u \in \mathcal{M}} u$
- Arc Relaxation: $\Gamma(a, \mathcal{M}) = v(a)$

3.1.3 Correctness

The correctness of the above model and its relaxation has been proved by Bergman et al. in [Ber+14b].

3.2 Maximum Cut Problem (MCP)

The Maximum Cut Problem (MCP) is another classic problem of graph theory which has – among other – been used in the context of very-large-scale integration design (VLSI) and statistical physics [Fes+02; CKC83; Pin84; CD87]. In essence, the problem consists in finding a bi-partition (S, T) of the vertices of some given graph that maximizes the total weight of edges whose endpoints are in different partitions. According to the appropriate terminology, the vertex partitioning is called a *cut* and the edges having their endpoints on either side of that cut are said to be *crossing the cut.* Thus, the MCP consists in finding a cut of maximum weight. That is, finding a cut maximizing the total weight of the edges crossing it.

Given an undirected weighted graph $G = (V, E)$ in which the weight of the edge $(i, j) \in E$ is denoted $w_{i,j}$, the MCP is expressed as the following optimization problem:

$$\max \sum_{i,j \in V} w_{i,j}(x_i \neq x_j) \tag{3.4}$$

$$x_i \in \{S, T\} \; \forall i \in V \tag{3.5}$$

In this formulation, a variable x_i is a binary variable indicating whether the vertex i is assigned to partition S or T (3.5). The objective function (3.4) simply computes the sum of the weight of edges crossing the S-T cut $(x_i \neq x_j)$.

3.2.1 DP Model

The simplest and most natural way of formulating a DP model for the MCP considers a DP state as the set of vertices belonging to one of the two partitions. In that case the transition function simply consists in the insertion (or not) of the given vertex v to the previous state. And the transition cost function adds the cost of the edges adjacents to v which are fully determined to cross the cut.

In [Ber+16b], Bergman et al. proposed a more elaborate model which eases the reconciliation of nodes having similar objective value. This is the model which is presented in the rest of this section. This model assumes that the graph $G = (V, E)$ be a complete graph, which has no impact on the generality of the method given that non existing edges can simply be assigned a 0 weight.

In this model, a state s^k from the kth layer is an n-tuple of integer whose component s^k_v indicates the marginal benefit of assigning vertex v to the partition T of the cut, based on the decisions made in the previous layers. Therefore, the initial and terminal states of this model are both of the form $\langle 0, 0, \ldots, 0 \rangle$

as the marginal benefit of moving any node to partition T is 0 when no decision has been made as well as when no decision can be made. In line with this representation, when a decision is made to assign vertex v to partition S, the transition function updates the marginal benefit of each remaining vertex u by adding the cost $w_{u,v}$ of the edge between these two vertices. Similarly, when v is added to partition T, the cost $w_{u,v}$ is subtracted from the marginal benefit of each remaining vertex u. Indeed, in that case $w_{u,v}$ is an opportunity cost that would be lost if u were added to T.

The subtlety of this model originates from its transition cost function. It would be incorrect for the latter to simply add s_v^v to the objective when a decision is made about v. As said previously, the s_v^v quantity represents the *marginal gain* of moving vertex v to partition T. Which means it is not a cost in its own right. It also means that this marginal benefit is only ever "gained" when the vertex is assigned in accordance with the sign of s_v^v; that is, when either $s_v^v > 0$ and v is assigned to T or $s_v^v < 0$ and v is assigned to partition S. Finally, the transition cost function also accounts for the opportunity cost of negative edges which explains the second term as well as the initial value of the DP model.

Formally the DP model is defined as follows. In this definition, the notations $(\alpha)^+$ is used as a shorthand for $\max\{0, \alpha\}$ and $(\alpha)^-$ stands for $\min\{0, \alpha\}$.

- State spaces: $S_k = \left\{s^k \in \mathbb{R}^n \mid s_j^k = 0, j = 0, \ldots, k\right\}$ with root and terminal states of the form $r = \langle 0, \ldots, 0\rangle$ and $t = \langle 0, \ldots, 0\rangle$
- State transition: $\tau_k(s^k, x_k) = (0, \ldots, 0, s_{k+1}^{k+1}, \ldots, s_{n-1}^{k+1})$ where

$$s_l^{k+1} = \left\{ \begin{array}{ll} s_l^k + w_{k,l} & \text{if } x_k = S \\ s_l^k - w_{k,l} & \text{if } x_k = T \end{array} \right\}, l = 1 + k, \ldots, n-1$$

- Transition cost: $h_0(s^0, x_0) = 0$ for $x_0 \in \{S, T\}$, and

$$h_k(s^k, x_k) = \left\{ \begin{array}{lll} (-s_k^k)^+ & + \sum_{\substack{l>k \\ s_l^k w_{k,l}<0}} \min\{|s_l^k|, |w_{k,l}|\}, & \text{if } x_k = S \\ (s_k^k)^+ & + \sum_{\substack{l>k \\ s_l^k w_{k,l}>0}} \min\{|s_l^k|, |w_{k,l}|\}, & \text{if } x_k = T \end{array} \right\},$$

$$k = 1, \ldots, n-1$$

- Root value: $v_r = \sum_{0 \le i < j < n} (w_{i,j})^-$

3.2.2 Relaxation

In their paper [Ber+16b], Bergman et al. propose the following relaxation of the above DP model. The merge operator is applied on a per-state component

basis. While it might seem as though a simple max operation should suffice to merge the state-components, the authors point out that it is actually not the case since it might cause a decrease in the length of a path passing through the merged node. Hence given that a valid relaxation mandates that all paths passing through a merged node must be at least as long as their exact counterpart, they proposed to proceed in two steps. First the least extreme value of each state-component is kept when all states agree on the direction of the incentive for that component (a positive value of s_i^k indicates an incentive to move vertex i to partition T, and a negative value an incentive to move it to partition S). The second step of their relaxation is operated by offsetting the potential losses on the arcs entering the merged node.

- Merge Operator: for any state-component $0 \leq l < n$,

$$\oplus(\mathcal{M})_l = \begin{cases} \min\limits_{u \in \mathcal{M}} \{u_l\} & \text{if } u_l > 0 \text{ for all } u \in \mathcal{M} \\ -\min\limits_{u \in \mathcal{M}} \{|u_l|\} & \text{if } u_l < 0 \text{ for all } u \in \mathcal{M} \\ 0 & \text{otherwise} \end{cases}$$

- Arc Relaxation $\Gamma(a, \mathcal{M}) = v(a) + \sum\limits_{0 \leq l < n} (|e(a)_l| - | \oplus (\mathcal{M})_l|)$

3.2.3 Correctness

The DP model we use for the MCP and its relaxation are those from [Ber+16b]. Their correctness is proved in the same paper.

3.3 Maximum 2-Satisfiability Problem (MAX2SAT)

MAX2SAT is a classic optimization problem at the border between operations research and logic. Over the years, it has – among other – been used in the context of planning, computer architecture design and Bayesian network learning [Ign+14; Cus08].

In propositional logic, a *literal* is either an atomic proposition (a boolean variable) or its negation. And while these literals can – in general – be combined with arbitrary logic connectives to form formulas, automated reasoning tools usually work with equisatisfiable formulas in conjunctive normal form (CNF) where all formula are expressed as *conjunction* of clauses [Tse68; BHM09]. That is, in CNF all formulas are expressed as conjunction of disjunction of literals. Given a logic formula in CNF whose clauses have each been assigned a weight, the MAXSAT problem consists in finding a variable assignment that maximizes the total weight of the satisfied clauses. MAX2SAT is a restricted form of the MAXSAT problem in which each clause of the formula

comprises at most two literals. Even though it might seem that MAX2SAT is easier to solve than the general MAXSAT case, it has been proved in [GJS74] that both problems are NP-hard.

Given a CNF formula bearing on a set $x = \{x_0, x_1, \ldots, x_{n-1}\}$ of boolean variables organized as m clauses, the MAX2SAT problem can be formally expressed as the following optimization problem.

$$\max \sum_{i=0}^{m} w_i c_i(x) \tag{3.6}$$

$$x_i \in \{T, F\} \, \forall 0 \leq i < n \tag{3.7}$$

In this formulation (3.6) is the objective to maximize where w_i is the weight associated to the ith clause and $c_i(x) = 1$ when clause i is satisfied and 0 otherwise. The constraint (3.7) enforces the binary domain of the problem variables.

3.3.1 DP Model

The DP model of MAX2SAT is very similar to that of MCP presented in section 3.2. In the same way the MCP model assumed – without loss of generality – the presence of all possible edges in the graph, the MAX2SAT model hypothesizes that any possible clause is present in the formula. Which can again be done without loss of generality since an absent clause can just as well be assumed to have a null weight for MAX2SAT purposes. Similar to the MCP case, the DP model for MAX2SAT also defines a state s^k as a tuple $\langle s_0^k, s_1^k, \ldots, s_{n-1}^k \rangle$ where each s_i^k component represents the marginal benefit of assigning the truth value T to variable x_i.

To better explain the behavior of the state transition function, let us start from an example. Considering the following problem comprising two clauses $(a \vee b)$ and $(\neg a \vee \neg b)$, both of which have an equal weight of 5.

$$\begin{array}{ccc|c} a & \vee & b & 5 \\ \neg a & \vee & \neg b & 5 \end{array}$$

There is no a priori reason to prefer the assignment $[\![b = T]\!]$ over $[\![b = F]\!]$. Indeed, both solutions $[\![a = T, b = F]\!]$ and $[\![a = F, b = T]\!]$ yield a total score of 10. However, as soon as the truth value T is assigned to a, F becomes the preferred value of b. Indeed, the assignment $[\![a = T, b = T]\!]$ only yields a meager score of 5 as opposed to the possible total of 10. The assignment $[\![a = T]\!]$ has thus decreased the marginal benefit of assignment $[\![b = T]\!]$ by 5 units. That alteration of the marginal benefit of $[\![b = T]\!]$ is due to the fact that when $[\![a = T]\!]$ the clause $(a \vee b)$ is satisfied regardless of the value of b. The

possibility of satisfying $(\neg a \vee \neg b)$ however depend on the value of b only. The transition function reflects that change in the marginal benefit of $[\![b = T]\!]$ by adding the potential gain of clause $(\neg a \vee b)$ and subtracting the opportunity cost of not satisfying $(\neg a \vee \neg b)$ from the marginal benefit of $[\![b = T]\!]$. In our example, the clause $(\neg a \vee b)$ is absent from the problem definition and is thus considered to be 0 weighted and $(\neg a \vee \neg b)$ has a weight of 5. Which is why the marginal benefit of $[\![b = T]\!]$ has become -5 after the assignment $[\![a = T]\!]$.

The reasoning would have been similar if a were to have been assigned the truth value F. But in that case, the clauses depending on b only would have been $(a \vee b)$ and $(a \vee \neg b)$. Thus, the marginal benefit would have been adapted according to the weight of these clauses.

The logic underlying the transition cost function of the MAX2SAT DP model is as follows. Upon branching on $[\![x_k = T]\!]$ from state s^k, the marginal benefit associated to $[\![x_k = T]\!]$ is immediately acquired if that benefit is positive (later denoted $(s^k_k)^+$). Otherwise that is an opportunity cost which has no direct impact on the global objective. Also, given that the decision $[\![x_k = T]\!]$ provokes the satisfaction of other clauses regardless the truth value assigned to the other literals, the cost of these clauses can be added to the objective ($\sum\limits_{l>k} w^{TT}_{k,l} + w^{TF}_{k,l}$ in the formal definition below). Beyond that, the transition cost function also incorporates a component to account for the minimum benefit which will arise from the assignment of the other literals for all the clauses depending on that second literal only ($\sum\limits_{l>k} \min \left\{ (s^k_l)^+ + w^{FT}_{k,l}, (-s^k_l)^+ + w^{FF}_{k,l} \right\}$). A symmetric reasoning is held upon branching on $[\![x_k = F]\!]$.

The initial value of this model simply consists of the sum of the weights of all tautological clauses. Indeed, these clauses are always satisfied – by definition of a tautology, but are not otherwise accounted for in the transition cost function.

3.3.1.1 Notations

The MAX2SAT DP model formalized hereafter uses the following notations:

- $(\alpha)^+$ as a shorthand notation for $\max\{0, \alpha\}$,
- $w^{TT}_{k,l}$ to denote the weight of the clause $(k \vee l)$,
- $w^{TF}_{k,l}$ to denote the weight of the clause $(k \vee \neg l)$,
- $w^{FT}_{k,l}$ to denote the weight of the clause $(\neg k \vee l)$ and
- $w^{FF}_{k,l}$ to denote the weight of the clause $(\neg k \vee \neg l)$.

3.3.1.2 Model

- State spaces: $S_k = \left\{ s^k \in \mathbb{R}^n \mid s^k_j = 0, j = 0, \ldots, k-1 \right\}$ with the root state and terminal state of the form $r = \langle 0, \ldots, 0 \rangle$ and $t = \langle 0, \ldots, 0 \rangle$.
- State transition: $\tau_k(s^k, x_k) = (0, \ldots, 0, s^{k+1}_{k+1}, \ldots, s^{k+1}_{n-1})$ where

$$s^{k+1}_l = \left\{ \begin{array}{ll} s^k_l + w^{TT}_{k,l} - w^{TF}_{k,l} & \text{if } x_k = F \\ s^k_l + w^{FT}_{k,l} - w^{FF}_{k,l} & \text{if } x_k = T \end{array} \right\}, l = k+1, \ldots, n-1$$

- Transition cost[1]: $h_0(s^0, x_0) = 0$ for $x_0 \in \{T, F\}$, and

$$h_k(s^k, x_k) = \left\{ \begin{array}{lll} (-s^k_k)^+ & +w^{FF}_{k,k} + \sum\limits_{l>k} (w^{FF}_{k,l} + w^{FT}_{k,l} + & \\ & \min \left\{ (s^k_l)^+ + w^{TT}_{k,l}, (-s^k_l)^+ + w^{TF}_{k,l} \right\}) & \text{if } x_k = F \\ & & \\ (s^k_k)^+ & +w^{TT}_{k,k} + \sum\limits_{l>k} (w^{TF}_{k,l} + w^{TT}_{k,l} + & \\ & \min \left\{ (s^k_l)^+ + w^{FT}_{k,l}, (-s^k_l)^+ + w^{FF}_{k,l} \right\}) & \text{if } x_k = T \end{array} \right\},$$

$$k = 1, \ldots, n-1$$

- Root value[2]: $v_r = \sum\limits_{i=0}^{n} w^{TF}_{i,i}$

3.3.2 Relaxation

The MAX2SAT model uses the exact same relaxation operators as MCP. Their interpretation is also similar. In both cases, the point is to reconcile (possibly divergent) marginal benefits.

3.3.3 Correctness

The DP model and relaxation we use for MAX2SAT again originates from [Ber+16b]. Their proof of correctness are to be found in the appendices of the same paper.

[1] This is a correction brought to the original DP formulation. Our model introduces the terms $w^{TT}_{k,k}$ and $w^{FF}_{k,k}$ in the transition cost function as a means to account for the cost of unit clauses. These were not originally accounted for in the model proposed by Bergman et al. The introduction of these terms does not fundamentally alters the reasoning behind the formal proof of correctness presented in the appendices of [Ber+16b].

[2] This is another correction brought to the original DP formulation which used a root value $v_r = 0$. This however, negatively impacted the value of the actual objective as the weight of tautological clauses was omitted. Apart from this change, and the introduction of terms to account for unit clauses in the transition cost function, no adaptation is required in the original model or in its proof of correctness.

3.4 Traveling Salesman Problem w. Time Windows (TSPTW)

TSPTW is a popular variant of the TSP where the salesman's customers must be visited within given time windows. The real world applications of this problem are obvious. For instance, the TSPTW could be immediately applicable to the scheduling of technicians appointment e.g. to activating DSL line. Other, less obvious uses have also been proposed for this problem. For instance [XQ02] proposes to apply it in the context of steel production.

In spite of being straightforward to express and comprehend, the TSPTW is notoriously hard to solve. Indeed, it turns out that even finding a feasible solution was proved NP-complete [Sav85]. Formally, TSPTW is characterized by N a number of customers to visit, $\mathcal{D}$ a square matrix s.t. $\mathcal{D}_{i,j}$ is the distance between customers i and j; $\mathcal{H}$ the considered time horizon and $\mathcal{TW}$ is a vector of time windows s.t. $\mathcal{TW}_i = (e_i, l_i)$ where e_i is the earliest time when the salesman can visit i and l_i the latest.

3.4.1 CP Model

The most natural way to formalize TSPTW is probably to express it as a CP model. Listing 3.1 presents a declarative Minizinc [Net+07] model for this problem. In this model, the decision variable x_i defines the visited customer in i^{th} position of the tour. The auxiliary variables a_i represent the time when the salesman visits the ith customer of the tour. The constraints ensure i) that the salesman's tour starts and ends at the depot ii) each city is visited exactly once iii) the time window constraints and iv) the salesman cannot travel faster than specified in the distance matrix between two consecutive visits but is allowed to wait until the beginning of the time window. Finally, the travel time objective is minimized.

```
1  /* Make it a tour start/ending at city 0 */
2  constraint (x[0] = 0 /\ x[N] = 0 /\ a[0] = 0);
3  constraint alldifferent_except_0(x);
4  /* Enforce time windows */
5  constraint forall(i in 0..N)(Earliest[x[i]] <= a[x[i]]);
6  constraint forall(i in 0..N)(a[x[i]] <= Latest[x[i]]);
7  constraint forall(i in 0..N-1)(
8    a[x[i+1]] >= a[x[i]] + Distance[x[i],x[i+1]]
9  );
10 /* Travel Objective */
11 int: travel = sum(i in 0..N-1)(Distance[x[i],x[i+1]]);
12 solve minimize travel;
```

Listing 3.1: Minizinc CP model for the TSPTW

3.4.2 DP Model

The DP model we use is a variation of the classical DP formulation that minimizes the travel time. In essence, the model remains exactly the same, only did we adapt the definition of a state to make it more amenable to merging.

Note

Because TSPTW is a *minimization* problem, it does not respect the usual DDO maximization assumption. Still, DDO remains perfectly usable. Indeed, it suffices to flip the sign of all costs of the transition cost function in order to turn a minimization problem into a maximization one.

Theorem 3.4.1 *The assignment minimizing function $f(x)$ is found as the assignment maximizing $-f(x)$. That is, by maximizing the objective function whose sign has been flipped.* $\min f(x) = \max -f(x)$

Proof 3.4.1 *Let us start with the obvious:*

$$f(x) = m \iff -f(x) = -m$$

Also, by definition of the maximum and minimum:

$$\begin{aligned}
\max f(x) = m &\iff \exists \alpha : f(\alpha) = m \wedge \forall \alpha' \neq \alpha : f(\alpha') \leq m \\
\min f(x) = m &\iff \exists \alpha : f(\alpha) = m \wedge \forall \alpha' \neq \alpha : f(\alpha') \geq m \\
&\iff \exists \alpha : -f(\alpha) = -m \wedge \forall \alpha' \neq \alpha : -f(\alpha') \leq -m \\
&\iff \max -f(x) = -m
\end{aligned}$$

■

In the context of DDO, flipping all the costs imposes that the costs on the transition cost function be negated and that the relaxation operation yields a *lower bound* on the objective value. That is, the negation of the relaxation must yield an upper bound in the corresponding maximization problem. Indeed, if the relaxed bound $\overline{f(x)}$ is a lower bound on the actual objective value $f(x)$, we have $\overline{f(x)} \leq f(x)$ and thus $-\overline{f(x)} \geq -f(x)$ which means $-\overline{f(x)}$ is an upper bound when $-f(x)$ is the objective to maximize.

Formally, a state s^k from the kth layer of the MDD is structured as a tuple $\langle position, time, must_visit, may_visit \rangle$ where *position* is the set of cities where the traveling salesman might possibly be after he visited the k first cities of his tour. As we will show later, this set is – by definition of the transition function – always a singleton except when s^k is the state of a merged node. The *time* component denotes the earliest time when the salesman could have arrived to some city $\in$ *position*. Finally, *must_visit* is the set of cities that

have not been visited on any of the $r - s^k$ path, and *may_visit* is the set of cities that have been visited along *some* of the $r - s^k$ paths but *not all of them.*

Given a state $s^k = \left\langle time^k, position^k, must_visit^k, might_visit^k \right\rangle$ and a next destination d_k, the state transition function simply reflects the new position and removes it from the set of places that *must* or *might* be visited in the rest of the tour ($must_visit^k$ and $might_visit^k$ respectively). It also adapts the time so as to account for the potential wait time before visiting customer d_k.

The transition cost function is quite straightforward as it simply yields the distance between the last known position ($position^k$) and the new one (d_k). In the particular case where $position^k$ is not a singleton set (that is, when a relaxed DD is being compiled and s^k is the result of a merge operation); the transition cost function must ensure to return an optimistic view on the travel time. This is why it returns $\min \left\{ \mathcal{D}_{i,d_k} \mid i \in position^k \right\}$.

The formal definition of the elements of our TSPTW DP model is the following:

- State spaces: The states space S_k of our DP model is the set of states $\left\langle position^k, time^k, must_visit^k, may_visit^k \right\rangle$ where $position^k$ is a subset of possible customers, $time^k$ indicates the earliest possible arrival at $position^k$. And where $must_visit^k$ and $might_visit^k$ are sets of cities. The initial state $r = \langle \{0\}, 0, \{0, \ldots, n-1\}, \emptyset \rangle$. All terminal states are of the form $\langle \{0\}, w, \emptyset, \emptyset \rangle$ with $0 < w \leq \mathcal{H}$.

- State transition:

  $$\begin{aligned} &\tau_k(\left\langle position^k, time^k, must_visit^k, might_visit^k \right\rangle, d_k) = \\ &\quad \left\langle position^{k+1}, time^{k+1}, must_visit^{k+1}, might_visit^{k+1} \right\rangle \end{aligned}$$

  where

  $$\begin{aligned} position^{k+1} &= \{d_k\} \\ time^{k+1} &= \max \left\{ e_{d_k}, time^k + \min \left\{ \mathcal{D}_{i,d_k} \mid i \in position^k \right\} \right\} \\ must_visit^{k+1} &= must_visit^k \setminus \{d_k\} \\ might_visit^{k+1} &= might_visit^k \setminus \{d_k\} \end{aligned}$$

  for $k = 0, 1, \ldots, n-1$.

- Transition cost function:

  $$\begin{aligned} &h_k(\left\langle position^k, time^k, must_visit^k, might_visit^k \right\rangle, d_k) = \\ &\quad \min \left\{ \mathcal{D}_{i,d_k} \mid i \in position^k \right\} \end{aligned}$$

- Root value: $v_r = 0$

3.4.3 Relaxation

Our relaxation leverages the structure of the states to perform a merge operation that loses as little information as possible. Our Γ operator is defined as the identity function: the cost of edges entering the merged node does not need to be altered. Our $\oplus$ operator, on the other hand, is slightly more complex. The result of a merge operation yields a state in which the salesman is considered to be in any of the places where he could potentially be. The current time is optimistically chosen as the minimum among the times of all nodes participating in the merge operation (3.9). Also, the set of cities that *must* be visited is restricted to the set of cities for which all the merged states agree (3.10). The set of cities that *might* be visited, on the other hand, accounts for all other possibilities (3.11 and 3.12). That is, the set of cities that might be visited comprises all cities that might have been visited from any of the states participating in the merge operation (3.11). In addition to these, the set of cities that are considered as potentially visitable also comprises all the cities for which a disagreement existed between at least two states participating in the merge (3.12). This is why in (3.12), the intersection of all cities that must be visited in all state is subtracted from the union of these sets of cities; leaving only those cities for which a disagreement exists. Formally, the merge operator is expressed as:

- Merge operator:

$$\oplus(\mathcal{M})_{position} = \bigcup_{a \in \mathcal{M}} position^a \tag{3.8}$$

$$\oplus(\mathcal{M})_{time} = \min\{time^a \mid a \in \mathcal{M}\} \tag{3.9}$$

$$\oplus(\mathcal{M})_{must_visit} = \bigcap_{a \in \mathcal{M}} must_visit^a \tag{3.10}$$

$$\oplus(\mathcal{M})_{might_visit} = \bigcup_{a \in \mathcal{M}} might_visit^a \tag{3.11}$$

$$\cup \left(\bigcup_{b \in \mathcal{M}} must_visit^b \setminus \bigcap_{c \in \mathcal{M}} must_visit^c \right) \tag{3.12}$$

- Arc relaxation: $\Gamma(a, \mathcal{M}) = v(a)$

3.4.4 Bibliographic Note

There is a long history of models and techniques to solve and approximate the TSPTW. In [CMT81], Christofides et al. proposed both a dynamic programming formulation of the problem and several state-space relaxations of

the problem. These differ from the one presented above since their goal is not to merge states with the intent of imposing limits on the memory consumption of a solver, but instead aim at mapping the complete state space onto a smaller one. In [CVH13], Ciré and Van Hoeve presented a structured approach to deriving bounds for common sequencing objective functions using relaxed MDDs. That approach is used to compute bounds on the travel time in the model presented above (Section 3.4.2). In the same paper, the authors proposed a *refinement heuristic* based on priorities assigned to the customers[3] to compile relaxed MDDs with tight bounds. Their intuition is that the bound derived from a relaxed DD would be tighter if the decision to visit the customers that are the farthest apart from all others led to an exact node in the DD. Even though that heuristic is intended for the case where relaxed DDs are compiled by separation, a selection heuristic for the top-down compilation case can easily be devised using the same intuition. This, however, has not been programmed in the code used to run the experiments of Chapters 4,5, and 6; all of which are based on the *MinLP* heuristic.

In [Sav85], Savelsbergh et al. proposed efficient feasibility checks which allowed to lift the Lin-Kernighan k-Opt move [LK73] from a local search resolution of the TSP to a local search resolution of the TSPTW. In [Dum+95], Dumas et al. introduced new rules to filter out infeasible solutions and thereby greatly improved the efficiency of DP-based solvers for the TSPTW. In essence, the check they propose is based on the existence of a partial order between the time windows in problem instances. From that partial order, it follows that some positions *must be* visited before others in all feasible solutions. While the details of that check have not been reproduced in these pages for the sake of brevity, it remains that the check proposed by Dumas et al. is not only compatible with the model we propose (and the further bounds presented in later chapters), but that check has been implemented in the `for_each_in_domain()` method of the model programmed to carry the experiments of Chapters 4,5 and 6. In 1996, Malandraki et al. proposed in [MD96] proposed a technique called *restricted dynamic programming* as a means to compute an upper bound on the optimal value of the Time Dependent Traveling Salesman Problem (TD-TSP): a problem which is close to the TSPTW. That very technique is essentially the same which has been described in section 2.3.4 of this manuscript for the compilation of restricted-MDD. More recently, Lopez-Ibañez et al. devised an heuristic method to solve TSPTW through a combination of beam search and ant-colony optimization [LIB10]. Along somewhat different lines, Baldacci et al. have investigated the use of ILP techniques to derive bounds on the optimal value of the TSPTW rather than DP [BMR12].

---
[3] jobs in the original paper

3.5 Pigment Sequencing Problem (PSP)

Intuitively, the Pigment Sequencing Problem (PSP) is a constrained planning problem where one wants to minimize the cost incurred by the production and stocking of goods while matching the imposed delivery date for these goods. The production is performed on a single machine that can be configured to produce any of the required items and produces one unit of it over the course of one atomic period of time. If an item is produced before its due date, then it must be stored. The stocking of an item incurs a cost which depends on the type of the stored item. Also, given that all items are produced on a single machine, the configuration of that machine must be adapted whenever the production needs to change. That configuration change has a cost which depends both on the current configuration and the next one. That is, it depends both on the type of the item which is currently produced and on the type of the next item to produce. This kind of situation would for instance arise if the production machine were a 3D printer whose nozzle, filament, bed and enclosure settings were to be adjusted depending on the prints.

More technically, the PSP is categorized as a multi-item capacited lot sizing problem. It is detailed in CSPLib [GW99, Problem 58] and studied in depth in [PW06]. It is formally characterized by a 5-tuple $\langle \mathcal{I}, \mathcal{H}, \mathcal{S}, C, Q \rangle$ where:

- $\mathcal{I} = \{0, \ldots, n-1\}$ is the set of item types to produce,
- $\mathcal{H}$ is the problem time horizon,
- $\mathcal{S}$ is a stocking cost vector where $\mathcal{S}_i$ is the cost of stocking one unit of type i during one period,
- C is a changeover cost matrix where $C_{i,j}$ is the cost of changing the machine configuration from producing item i to producing item j, and
- Q is a vector of demands per item. Given a time period $0 \leq t < \mathcal{H}$ and an item $i \in \mathcal{I}$, Q_t^i is used to denote the number of items of type i to deliver at time t. Without loss of generality, the rest of this manuscript assumes normalized demands. That is, $Q_t^i \in \{0, 1\}\ \forall t, i$.

On that base, the PSP can be expressed as shown per equations (3.13) – (3.18). In line with the nomenclature from [PW06], this model will be referred to as PIG-A-1 in the rest of this thesis.

$$\textbf{minimize} \sum_{i,j,t} C_{i,j} \mathbf{c}_{i,j}^t + \sum_{i,t} \mathcal{S}_i \mathbf{s}_i^t \tag{3.13}$$

subject to

$$\mathbf{s}_i^0 = 0 \;\; \forall i \in \mathcal{I} \tag{3.14}$$

$$\mathbf{x}_i^t + \mathbf{s}_i^{t-1} = Q_t^i + \mathbf{s}_i^t \;\; \forall i \in \mathcal{I}; 0 \leq t < \mathcal{H} \tag{3.15}$$

$$\mathbf{x}_i^t \leq \mathbf{y}_i^t \;\; \forall i \in \mathcal{I}, 0 \leq t < \mathcal{H} \tag{3.16}$$

$$\sum_{i,t} \mathbf{y}_i^t = 1 \;\; \forall i \in \mathcal{I}, 0 \leq t < \mathcal{H} \tag{3.17}$$

$$\mathbf{c}_{i,j}^t \geq \mathbf{y}_i^{t-1} + \mathbf{y}_j^t - 1 \;\; \forall i, j \in \mathcal{I}; 0 \leq t < \mathcal{H} \tag{3.18}$$

In the above model, $\mathbf{x}_i^t$ is a binary production variable (1 when item i is produced at time t, otherwise 0). $\mathbf{y}_i^t$ is a binary setup variable (1 iff machine is ready to produce i at time t). $\mathbf{c}_{i,j}^t$ is a binary changeover variable (1 iff configuration changed from i to j at time t). $\mathbf{s}_i^t$ is an integer socking variable counting the number of items of type i stored at time t.

The equation (3.13) expresses the objective function to minimize. Also, the constraint (3.14) imposes that the stock of every item is empty at startup. Equation (3.15) is a *conservation* constraint stating that when an item is produced, it is either delivered or stocked for later delivery. Constraint (3.16) forces the consistency between the production and machine configuration variables. (3.17) is a *capacity* constraint stating that only 1 unit of one item is produced at each time. Finally, (3.18) is a constraint that enforces the consistency between the machine configuration variables ($\mathbf{y}_i^{t-1}, \mathbf{y}_j^t$) and the changeover variables ($\mathbf{c}_{i,j}^t$).

3.5.1 DP Model

Note

To the best of our knowledge, this problem has not been solved with DP before. The model we proposed in [GS22] and which we hereby reproduce should therefore be considered a contribution of this thesis.

The DP model we propose for the PSP works as follows: the decisions that are made should be interpreted as answering the question "what item is produced at time t ?". For example, a decision $[\![x_4 = 3]\!]$ would mean that item 3 is to be scheduled for production on the machine at time 4. Also, our model starts by making decision about the items that must be produced last (at time $\mathcal{H}$) progressing towards the start of the plan. This approach has been chosen to avoid branching on infeasible paths during the compilation of the DDs.

In this model, a state is a tuple $\langle k, u \rangle$ where k can either be an item from the item set $\mathcal{I}$ or -1. Intuitively, the k component of a state $s^t = \langle k, u \rangle$ is meant to stand for the item that will be scheduled for production at time $t+1$.

The u component is a vector indicating for each item, the time at which the previous demand had to be satisfied.

Obviously, given that the problem does not consider any production after the time horizon $\mathcal{H}$, the component k can also hold a 'dummy' value. We used value -1 to materialize this dummy value. In the definitions below, a state where $k = -1$ should be interpreted as a means to signify that any item could have been scheduled for production at time $t+1$. The dummy value -1 can also occur in the context of the transition and transition cost functions. In that case however, a decision to branch on item -1 should be understood as a decision to leave the machine idle for one time step.

The transition function proceeds as follows: when the machine is left idle, the state is left untouched (obviously). Otherwise, when an actual item is scheduled for production, the identifier of this item is stored in the k state component and its previous demand time is updated. Similarly, the transition cost function yields a cost of zero when the machine is left idle. When an actual item is scheduled for production, it returns a cost that accounts for both the machine reconfiguration (which can amount to zero when $k = -1$) and the stocking cost of the item that is being scheduled.

Formalization In order to formally define a DP model for the PSP, it helps to first define $\mathcal{T}_t^i$ from the input data, which yields the *previous* demand time for a given item i and time $0 \le t \le \mathcal{H}$.

$$\mathcal{T}_0^i = -1 \quad \Big| \quad \mathcal{T}_t^i = \begin{cases} t-1 & \text{if } Q_i^{t-1} > 0 \\ \mathcal{T}_{t-1}^i & \text{otherwise} \end{cases}$$

The elements of a PSP DP model are the following:

- a state $s^t \in \mathcal{S}_{\mathcal{H}-t}$ is a tuple $\langle k, u \rangle$ where k denotes the type produced at time $t+1$, and u is a vector comprising the previous delivery date for each item. In particular, we have $r = \left\langle -1, \left(\mathcal{T}_{\mathcal{H}}^0, \mathcal{T}_{\mathcal{H}}^1, \dots, \mathcal{T}_{\mathcal{H}}^{n-1}\right)\right\rangle$

- $$\tau_t(\langle k, u \rangle, d) = \begin{cases} \langle k, u \rangle & \text{when } d = -1 \\ \left\langle d, \left(u_0, \dots u_{d-1}, \mathcal{T}_{u_d}^d, u_{d+1}, \dots u_{n-1}\right)\right\rangle & \text{when } u_d \ge t \\ \bot & \text{otherwise} \end{cases}$$

- $$h_t(\langle k, u \rangle, d) = \begin{cases} \mathcal{S}_d \cdot (u_d - t) & \text{when } k = -1 \\ C_{k,d} + \mathcal{S}_d \cdot (u_d - t) & \text{otherwise} \end{cases}$$

- $v_r = 0$.

3.5.2 Relaxation

Note

Similar to TSPTW, the PSP is a *minimization* problem. Its correctness therefore depends on its ability to derive *lower bounds* from relaxed nodes.

Based on the above DP model, a simple and correct way of merging several nodes consists in considering that the item that has been scheduled for production at time $t + 1$ could be any item at all. Moreover, our merge operator keeps the smallest previous demand time for all items. From this choice, it results that some items might "disappear" from the merge state, thereby forcing the machine to remain idle for one or more time periods.

- Merge operator: $\oplus_t(\mathcal{M}) = \left\langle -1, \left(\min_{a \in \mathcal{M}} u_0^a, \min_{a \in \mathcal{M}} u_1^a, \ldots, \min_{a \in \mathcal{M}} u_{n-1}^a \right) \right\rangle$
- Arc relaxation: $\Gamma(a, \mathcal{M}) = v(a)$

3.6 Description of the benchmark instances

This section describes for each problem the benchmark instances that will be used in the experimental studies of following chapters.

MISP. In order to evaluate the impact on MISP of the various techniques proposed throughout this thesis, we generated random graphs based on the Erdos-Renyi model G(n, p) [ER59] with the number of vertices n = 250, 500, 750, 1000, 1250, 1500, 1750 and the probability of having an edge connecting any two vertices p = 0.1, 0.2, ... , 0.9. The weight of the edges in the generated graphs were drawn uniformly from the set $\{-5, -4, -3, -2, -1, 1, 2, 3, 4, 5\}$. We generated 10 instances for each combination of size and density (n, p).

MCP. In line with the strategy used for MISP, we generated random MCP instances as random graphs based on the Erdos-Renyi model G(n, p). These graphs were generated with the number of vertices n = 30, 40, 50 and the probability p of connecting any two vertices = 0.1, 0.2, 0.3, .., 0.9. The weights of the edges in the generated graphs were drawn uniformly among $\{-1, 1\}$. Again, we generated 10 instances per combination n, p.

MAX2SAT. Similar to the above, we used random graphs based the Erdos-Renyi model G(n, p) to derive MAX2SAT instances. To this end, we produced graphs with n = 60, 80, 100, 200, 400, 1000 (hence instances with 30, 40, 50,

100, 200 and 500 variables) and p = 0.1, 0.2, 0.3, .. , 0.9. For each combination of size (n) and density (p), we generated 10 instances. The weights of the clauses in the generated instances were drawn uniformly from the set $\{1, 2, 3, 5, 6, 7, 8, 9, 10\}$.

TSPTW. To evaluate the effectiveness of our rules on TSPTW, we used the 467 instances from the following suites of benchmarks, which are usually used to assess the efficiency of new TSPTW solvers. AFG [Asc96], Dumas [Dum+95], Gendreau-Dumas [Gen+98], Langevin [Lan+93], Ohlmann-Thomas [OT07], Solomon-Pesant [Pes+98] and Solomon-Potvin-Bengio [PB96]. All these benchmark instances as well as their best known solutions are available online at [LIB20].

PSP. In order to evaluate the effectiveness of the techniques that are proposed in this manuscript, we generated two sets of random PSP instances with varying number of items and time horizon. Table 3.1 shows the details of the configurations for the first set of benchmarks instances. It comprises 500 generated instances with 5 items ($|\mathcal{I}| = 5$) and a time horizon varying between 20 and 100. The details of the second set of benchmark instances is shown in table 3.2. This data set comprises 500 generated instances having 10 items and a time horizon varying between 50 and 100.

The strategy which has been used to generate the stocking and changeover costs of the random instances was the same for both data sets. In bot cases, the stocking and changeover costs have been generated so as to reflect a variety of possible situations: stocking cost dominated, changeover dominated, balanced between the costs.

features	number of instances
$\lvert\mathcal{I}\rvert = 5, \mathcal{H} = 20$	100
$\lvert\mathcal{I}\rvert = 5, \mathcal{H} = 50$	100
$\lvert\mathcal{I}\rvert = 5, \mathcal{H} = 100$	300

Table 3.1: Details of the 1st set of benchmark instances

features	number of instances
$\lvert\mathcal{I}\rvert = 10, \mathcal{H} = 50$	300
$\lvert\mathcal{I}\rvert = 10, \mathcal{H} = 100$	200

Table 3.2: Details of the 2nd set of benchmark instances

How to Implement a Fast and Generic DDO solver ? | 4

Contributions and Publication Information

This chapter substantially extends the content that has been published in Xavier Gillard, Pierre Schaus, and Vianney Coppé. *Ddo, a generic and efficient framework for MDD-based optimization.* IJCAI. 2020. Its main contribution comes in the form of *ddo*: the first generic open source library to implement fast and efficient solvers based on the branch and bound with DD paradigm. This chapter complements the original publication with a discussion on the engineering and performance aspects of implementing *ddo*.

This chapter presents *ddo*, a generic and efficient library to solve constraint optimization problems with decision diagrams. This framework implements the branch-and-bound approach presented in Chapter 2 and provides a common ground to easily program efficient solvers based on the DDO paradigm. In particular, this single generic library allowed us to write solvers for the MISP, MCP, MAX2SAT, TSPTW and PSP, which reduced the effort of implementing all five problems to a mere translation of the models presented in Chapter 3. Besides that, this chapter discusses engineering aspects that are key to the performance of solvers developed with such a library. More specifically, it discusses four alternative implementations of the MDD abstraction and compares their respective performance with a computational study.

4.1 The *ddo* library

One of the key "selling point" of DDO as a combinatorial optimization technique is its simplicity. Indeed, all what is required in order to solve a problem using the branch-and-bound with DD algorithm is a DP model for the problem and a relaxation. Our goal while implementing our library was to keep this simplicity and to make it center and front. This is why the only two ingredients required from anyone willing to solve a new of problem using our library are a DP formulation and a suitable relaxation for the problem. In an ideal world these would be the only two inputs. Still, extensibility and con-

figurability have been thought from the start. This way, new data structures and heuristics can be provided by the user if deemed useful.

Given the relative youth of DDO as a sub-field in Operations Research, it has not gained a very wide adoption (yet), as opposed to the well established MIP for instance. By releasing an easy-to-use, free and efficient DDO library, we hoped to increase the public awareness of this field and lower its barrier to entry. This is why *ddo* was released as an open source[1] Rust library (crate in Rust parlance) alongside with its companion example programs to solve the aforementioned problems. To the best of our knowledge, this is the first public implementation of a generic library to solve combinatorial optimization problems with branch-and-bound MDD.

4.1.1 Why Rust ?

Considering that this project was started from scratch and did not have to deal with any kind of legacy code, we were faced with a complete technological choice freedom. After experimenting a bit with implementations in Java, Scala and C++, the choice finally settled on using Rust to write this library.

Rust[Fou22] is a recent programming language when compared to Java, Python or C++. It first appeared in 2010 at Mozilla Research in the context of the Servo project (Mozilla's experimental browser engine). Its purpose was to enable the development of key components of the browser while increasing the code readability, reducing the risk of memory related issues and keeping the resulting binary portable and extremely fast. Since then, the language has left the Mozilla-only scope and is now managed by the Rust Foundation. Over time, it started to gain corporate support[2] and it has been elected "the most loved programming language" in the Stack Overflow Developer Survey every year since 2016[Ove22].

Because of its absence of runtime (no virtual machine, no garbage collector) Rust is often described as a *low level system programming language*. While this is certainly true and embedded projects are being developed in Rust, the Foundation advertises it as "A language empowering everyone to build reliable and efficient software"[Fou22]. This description better encompasses our experience with the language and the reasons that motivated the technological choice behind *ddo*. Indeed, Rust seems to have brought about the benefits from all major languages without their inherent drawbacks. For instance, it features a strong type system structured with *traits* and implementation types similar to Java. Its tooling is also very comfortable to work with. For instance, unit tests documentation and doctests are integral to the language. Its build manager `cargo` will run them upon compilation for early problem

[1]https://github.com/xgillard/ddo
[2]https://www.rust-lang.org/sponsors

detection. Moreover, `cargo` comes with dependency management and build configuration capabilities similar to those of `maven` in the Java world. Besides that, Rust gives the developer a fine control over how the memory is used (stack allocated vs heap allocated, references and raw pointers) as is the case in C and C++. Combining that with the sheer performance of the generated binary code proved extremely valuable to developing *ddo* since solvers based on branch-and-bound with MDD paradigm are both CPU and memory intensive processes.

Finally, the built-in borrow checker[3] helped us a great deal in developing the parallel computing capabilities of this library. The strict rules it enforces[4] helped to prevent many of the issues typically related to this kind of development.

4.1.2 Parallel Computing

In [Ber+14c], Bergman et al. have shown that one of the strengths of the whole branch-and-bound with MDD approach was its ability to exploit the parallel computing capabilities of modern hardware. Our *ddo* library was therefore developed with parallelism in mind, building on the excellent tooling offered by the Rust ecosystem. In fine, *ddo* is able to exploit all the available hardware (or just a fraction of it) without imposing any constraint[5] on the person writing a solver for a specific problem.

Algorithms 6, 7, and 8 detail the pseudocode of how such a parallel solver can be programmed – which closely matches the parallel solver implementation in *ddo*. Per se, Algorithm 6 does very little. On the face of it, it exposes the same interface as the sequential Branch-and-Bound with MDD algorithm (Algorithm 5 from Chapter 2, p. 18). It requires the same DP model and relaxation operators inputs and yields the same output: a 2-tuple whose first term is an optimal solution to the problem and the second term is the objective value of that optimal solution. The major difference between the two algorithms stems from the fact that Algorithm 6 offloads all of the intensive computation to worker threads which collaborate to find the optimal solution (lines 13–20). Apart from that, the second key piece of information to take away from Algorithm 6 is that even though all threads share a common zone of memory, that zone is split into two distinct portions. The first one is

[3] The borrow checker is a utility integrated to the Rust compiler. It verifies during the compilation that at any time during the execution of the compiled code, no two distinct mutable references exist to the same memory location; and no variable is used after it has been freed (no dangling references). That property is used to guarantee the absence of race conditions in the code *at compile time*. A peculiarity of the Rust compiler is that any failure to comply with the rules enforced by the borrow checker results in a compilation failure.

[4] These can be locally overruled with `unsafe` Rust.

[5] Apart from memory safety.

the *shared* portion: it consists of all the (immutable) problem inputs and can safely be accessed by all threads concurrently. The second portion is called the *critical* portion and contains the mutable fraction of the solver state. Because of that mutable nature, and to avoid the occurrence of *data races*, all access to the critical portion must happen inside of the solver critical sections. In other words, all accesses to the critical data must be guarded by the acquisition of a mutex lock guaranteeing the synchronization at these specific points. In Algorithms 7 and 8, the critical sections have been visually highlighted with a darker background.

Algorithm 6 Parallel version of the Branch-and-Bound with MDD

```
1:  Input: a DP-Model 𝒫 = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  Input: a node merging operator ⊕
3:  Input: an arc relaxation operator Γ
4:  // Create the critical portion of the solver state (Crit) that can only be accessed
5:  // from the critical sections
6:  Mutex ← new Mutex              // Controls the access to the critical sections
7:  Crit.Fringe ← new Fringe       // The global solver fringe
8:  Crit.Ongoing ← 0               // Number of nodes currently being processed
9:  Crit.BestLB ← −∞               // The best lower bound known so far
10: Crit.BestSol ← ⊥               // The best solution known so far
11:
12: Crit.Fringe.add(r)             // Enqueue the root node of the problem
13:
14: // Start all worker threads
15: for 0 ≤ i < NbThreads do
16:     Spawn Thread(𝒫, ⊕, Γ, Mutex, Crit) with id i
17:
18: // Wait until all threads have finished
19: for 0 ≤ i < NbThreads do
20:     Wait for completion of Thread with id i
21:
22: // As in the sequential case, return the best solution and its value
23: return (Crit.BestSol, Crit.BestLB)
```

The reader familiar with the sequential version of the branch-and-bound with MDD algorithm (Algorithm 5, p. 18 in Chapter 2) will easily recognize that, for the most part, Algorithm 7 reproduces the content of the sequential algorithm (lines 13–28). The most noticeable difference is to be observed at lines 8 to 12. Indeed, rather than just popping a node off the solver Fringe as was the case in Algorithm 5, the parallel version calls the `GetWorkItem` procedure (detailed in Algorithm 8, p. 45). That procedure returns a 2-tuple

whose first term is a status $\in$ $\{Complete, Starvation, WorkAssigned\}$, and whose second term is an optional node popped off the solver Fringe. The latter is only populated when the returned status is $WorkAssigned$. The other two possible statuses serve the purpose of detecting special conditions like the termination ($status = Complete$) which is slightly harder to detect when parallelism is involved than in the sequential case, and a starvation condition ($status = Starvation$) which occurs when the solver is temporarily unable to provide a thread with some work as the Fringe is momentarily empty. Another important – albeit somewhat less noticeable – difference between the body of the sequential and parallel algorithm is visible at line 29. That seemingly innocuous line decrements a counter of active threads. That operation serves two purposes: first, it notifies the solver that the current thread is done with the processing of the subproblem it was assigned and the subproblems that had to be enqueued have been pushed onto the Frontier. The second purpose of decrementing the value of that counter is to help to detect termination when optimality of the best known solution has been proved.

As explained above, the `GetWorkItem` procedure detailed in Algorithm 8 serves two purposes: first it is used to detect the current status of the overall resolution. That status can either be:

- `Complete` when the search is complete and the solution stored in $Crit.BestSol$ is provably optimal. This condition happens when the Fringe is empty *and* there are no active worker theads left ($Crit.Ongoing = 0$).
- `Starvation` when the search is not complete but the solver is momentarily incapable of assigning work to the idle threads. That condition occurs when the Frontier is empty but some other workers are actively processing subproblems ($Crit.Ongoing > 0$).
- `WorkAssigned` when the Fringe contains subproblems that must be processed.

In the event where the status is `WorkAssigned`, the `GetWorkItem` procedure eagerly removes one node from the solver Fringe and assigns it to the current worker thread. In that case, it also increments the counter $Crit.Ongoing$ of active workers to signify that the current thread has officially been charged of exploring that particular subproblem.

4.1.3 Discussion

As shown per Algorithms 6, 7, and 8; *ddo* adopts a coarse grained parallelism model where each thread respectively handles the complete lifecycle of a node that was popped out of the solver Frontier. Another option would have been to opt for a fine grained parallelism where the expansion of several nodes is

Algorithm 7 Body of one worker thread

```
1:  Input:  a DP-Model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  Input:  a node merging operator ⊕
3:  Input:  an arc relaxation operator Γ
4:  Input:  Mutex a mutex controlling the access to the shared memory
5:  Input:  Crit a shared memory only be accessible in critical sections
6:  // Repeatedly try to get a subproblem to process until there is no more work to do
7:  for ever do
8:      ⟨Status, u⟩ ← GetWorkItem(Mutex, Crit)
9:      if Status is Complete then
10:         return      // No more work to do
11:     else if Status is Starvation then
12:         Wait until starvation is over (typically using monitor)
13:     else
14:         B̲ ← Restricted(u)
15:         // First Critical Section
16:         Mutex.lock()
17:         if v*(B̲) > Crit.BestLB then
18:             Crit.BestLB ← v*(B̲)
19:             Crit.BestSol ← x*(B̲)
20:         Mutex.unlock()
21:
22:         if B̲ is not exact then
23:             B̄ ← Relaxed(u, ⊕, Γ)
24:             // Second Critical Section
25:             Mutex.lock()
26:             if v*(B̄) > Crit.BestLB then
27:                 for all u′ ∈ B̄.exact_cutset()  do
28:                     Crit.Fringe.add(u′)
29:             Crit.Ongoing ← Crit.Ongoing − 1
30:             Mutex.unlock()
```

Algorithm 8 GetWorkItem procedure

Input: $Mutex$ a mutex controlling the access to the shared memory
Input: $Crit$ a shared memory only be accessible in critical sections
$Status \leftarrow$ Complete
$u \leftarrow None$

$Mutex.lock()$
if $Crit.Fringe$ is empty and $Crit.Ongoing = 0$ **then**
  $Status \leftarrow$ Complete
else if $Crit.Fringe$ is empty **then**
  $Status \leftarrow$ Starvation
else
  $Status \leftarrow$ WorkAssigned
  $u \leftarrow Crit.Fringe.pop()$
  $Crit.Ongoing \leftarrow Crit.Ongoing + 1$
$Mutex.unlock()$

return $\langle Status, u \rangle$

done in parallel during the compilation of DDs. This second approach was discarded because of the high amount of synchronization overhead it would involve compared to the coarse grained approach. As a matter of fact, the *ddo* implementation is almost lock free. The only three *mandatory* synchronization points being the moment where a node is popped out of the Fringe (Algorithm 8), the moment when an improved solution must be remembered (Algorithm 7 line 19), and when the nodes from the exact cutset are pushed on it (line 28). In addition to these, *ddo* implements a fourth synchronization point between the compilation of restricted and relaxed DD (in the pseudocode, that synchronization point would occur right before line 23 of Algorithm 7). That fourth synchronization point allows a running thread to make use of potentially better bounds that could have been derived in other threads[6].

It is worth mentioning that the parallel branch-and-bound with MDD algorithm depicted above does not require that one resort to using some advanced load balancing or work stealing heuristics. Indeed, the heaviest computation made by each thread is the compilation of the restricted and relaxed decision diagrams (Algorithm 7 lines 14 and 23). *Assuming that the complexity of both the transition and transition cost functions are* $O(1)$, the complexity of these compilations is $O(nWd)$ where n is the number of decision variables and W is the maximum layer width, and d the size of the largest variable domain in the problem. While these computations are expensive, the time they

[6] e.g. Using the techniques presented in the next chapters

take is well bounded and not subject to divergence. This means each thread will regularly "be freed" and post new nodes from a fresh exact cutset onto the Fringe (Algorithm 7 line 28). The time spent in the critical sections of Algorithms 7, and 8 is bounded by the complexity of adding a node to the Fringe and of popping a node off of it. Assuming that the Fringe was implemented using a binary heap, the complexity of the insertion of one node to the Fringe is $O(log_2 n)$ where n is the size of that Fringe. Likewise, the complexity of popping a node off the Fringe is $O(log_2 n)$ as well. Given that the second critical section of Algorithm 7 does not add a single node, but a complete cutset; the time spent in the critical sections of Algorithms 7 and 8 is bounded by $O(W log_2 n)$ since the size of an exact cutset cannot exceed the limit W. Hence the time spent in the critical sections grows slowly compared to the compilation of the decisions diagrams, even when the Fringe contains a reasonably large number of nodes. This explains why the compilation of the decision diagrams dominates the time spent in the critical sections in practice.

Also, because W is typically much larger than the number of CPU cores, chances are that whenever a worker thread pushes nodes onto the Frontier, it inserts more nodes than the number of workers. Therefore, when a thread requests a new subproblem to process, it is thus likely that the Frontier will be far from empty.

In practice, we observed very little contention during our experiments. The starvation condition occurred mostly near the complete resolution of a problem. Which is why the number of DDs compiled by second increased roughly linearly when using this parallelization scheme. This has been observed in experiments using as many as 40 threads running on a server equipped with two Intel Xeon E5-2687W v3 processors.

4.2 Design of the *ddo* library

Figure 4.1 depicts a high-level UML class diagram of the overall structure of the library. The coming Sections 4.2.1 and 4.3 describe in more detail the role and organization of the structures and interfaces (traits) depicted in that figure. As a first overview, let us first observe that the architecture of *ddo* was designed with modularity and loose coupling in mind. This is why there are very few structures that are fixed and can't be swapped for other implementations. In Figure 4.1, these are pictured with a yellow-shade background. The role of these objects is to establish a common protocol to exchange data between the different parts of the library. It should also immediately be apparent that the `Problem` and `Relaxation` traits (orange) are both very close to the mathematical definition given to these concepts in Chapter 2 and central to the design of the library.

In Figure 4.1 one can also see that the interface of a `Solver` is fairly simple

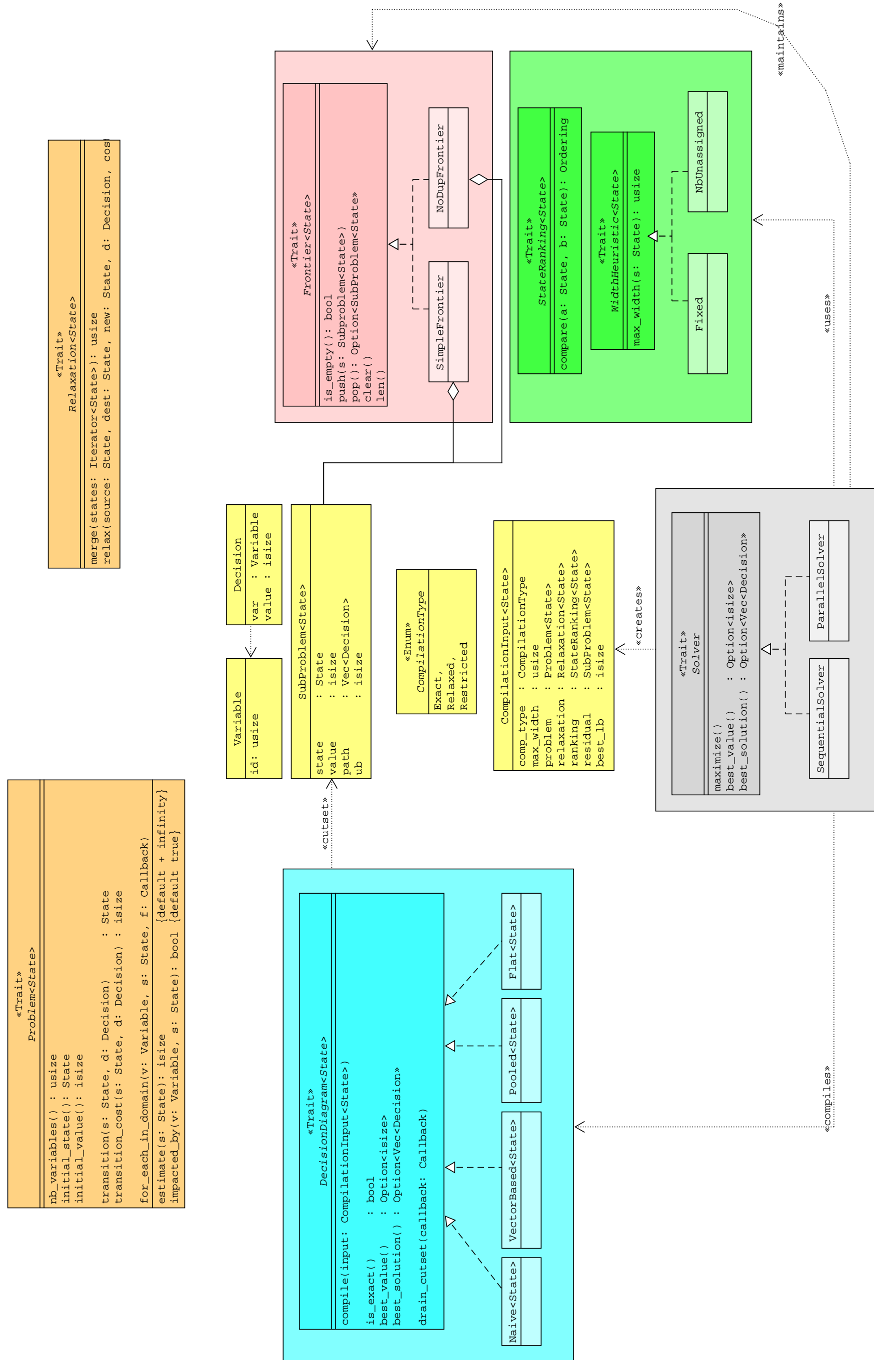


Figure 4.1: High-level UML class diagram of the overall *ddo* architecture

(light gray background). But the responsibilities of the actual solver implementations are very important: they are in charge of maintaining the frontier of open subproblems (pink background) while compiling some approximate decision diagrams (cyan) using the set of provided heuristics (green) to derive as tight bounds as possible.

4.2.1 A Few Key Abstractions

In line with the mathematical foundations of DDO – which we presented in Chapter 2, the implementatin of our *ddo* library relies on a few key abstractions. The simplest among these are probably those defining the representations of a `Variable` and a `Decision`. As shown per listing 4.1, a `Variable` is implemented as a mere type-safe unsigned integer. Such `Variables` are intended to be used as opaque identifiers where `Variable(i)` essentially denotes the variable x_i from the abstract mathematical models. Similarly, a `Decision` simply represents an assignment decision that was made by the solver. Put another way, it represents a choice that was taken to set a specific value to some given variable. For example, the assignment decision $[\![x_3 = 4]\!]$ is represented by `Decision{var: Variable(3), value: 4}`.

```
1  /// In essence, a Varible is nothing but a type-safe unsigned integer. It is used
2  /// as an unique identifier for some variable x_i in the problem to solve.
3  #[derive(Debug, Clone, Copy)]
4  pub struct Variable(pub usize);
5
6  /// A Decision basically represents a choice which is made to assign some
7  /// value (in this case a signed integer) to a given variable
8  #[derive(Debug, Clone, Copy)]
9  pub struct Decision {
10     pub var: Variable,
11     pub value: isize,
12 }
```

Listing 4.1: Variable and Decision, the simplest abstractions

As explained earlier, the two ingredients which are central to solving a combinatorial optimization problem with DD-based branch-and-bound are respectively: a DP model of the problem at hand, and a relaxation for that specific problem. Naturally, these elements are of utmost importance in our library as well. As shown per listing 4.2, the concepts for the DP model and its relaxation are defined as traits in *ddo*, and the operations they define as well as their semantic closely match their equivalent in the mathematical model presented in Chapter 2. For instance, in the definition of a problem, the transition and transition cost functions τ and h correspond to the method `tansition()` and `transition_cost()`. Similarly, the operators $\oplus$ and Γ which have been discussed in section 2.3.5 correspond to the methods `merge()` and `relax()` from the `Relaxation` trait.

```
/// This trait defines the "contract" of what defines an optimization problem
/// solvable with the branch-and-bound with DD paradigm. An implementation of
/// this trait effectively defines a DP formulation of the problem being solved.
/// That DP model is envisioned as a labeled transition system -- which makes
/// it more amenable to DD compilation.
pub trait Problem {
    /// The DP model of the problem manipulates a state which is user-defined.
    /// Any type implementing Problem must thus specify the type of its state.
    type State;
    /// Any problem bears on a number of variable x_0, x_1, x_2, ..., x_{n-1}
    /// This method returns the value of the number n
    fn nb_variables(&self) -> usize;
    /// This method returns the initial state of the problem (the state of r).
    fn initial_state(&self) -> Self::State;
    /// This method returns the intial value v_r of the problem
    fn initial_value(&self) -> isize;
    /// This method is an implementation of the transition function mentioned
    /// in the mathematical model of a DP formulation for some problem.
    fn transition(&self, state: &Self::State, decision: Decision) -> Self::State;
    /// This method is an implementation of the transition cost function mentioned
    /// in the mathematical model of a DP formulation for some problem.
    fn transition_cost(&self, state: &Self::State, decision: Decision) -> isize;
    /// Any problem needs to be able to specify an ordering on the variables
    /// in order to decide which variable should be assigned next. This choice
    /// is an **heuristic** choice. The variable ordering does not need to be
    /// fixed either. It may depend on the nodes constitutive of the next layer.
    /// These nodes are made accessible to this method as an iterator.
    fn next_variable(&self, next_layer: &mut dyn Iterator<Item = &Self::State>)
        -> Option<Variable>;
    /// This method calls the function `f` for any value in the domain of
    /// variable `var` when in state `state`.  The function `f` is a function
    /// (callback, closure, ..) that accepts one decision.
    fn for_each_in_domain<F>(&self, var: Variable, state: &Self::State, f: F)
    where F: FnMut(Decision);
}

/// A relaxation encapsulates the relaxation Γ and ⊕ which are
/// necessary when compiling relaxed DDs. These operators respectively relax
/// the weight of an arc towards a merged node, and merges the state of two or
/// more nodes so as to create a new inexact node.
pub trait Relaxation {
    /// Similar to the DP model of the problem it relaxes, a relaxation operates
    /// on a set of states (the same as the problem).
    type State;

    /// This method implements the merge operation: it combines several `states`
    /// and yields a new state which is supposed to stand for all the other
    /// merged states. In the mathematical model, this operation was denoted
    /// with the ⊕ operator.
    fn merge(&self, states: &mut dyn Iterator<Item = &Self::State>) -> Self::State;
    /// This method relaxes the cost associated to a particular decision. It
    /// is called for any arc labeled `decision` whose weight needs to be
    /// adjusted because it is redirected from connecting `src` with `dst` to
    /// connecting `src` with `new`. In the mathematical model, this operation
    /// is denoted by the operator Γ.
    fn relax(
        &self,
        source: &Self::State,
        dest: &Self::State,
        new: &Self::State,
```

```
        decision: Decision,
        cost: isize,
    ) -> isize;
}
```

Listing 4.2: The central ingredients of a ‘ddo‘ solver

In addition to the above, our *ddo* library also defines the following traits: `StateRanking` which encapsulates the heuristic selection from line 6 in algorithm 4. That heuristic imposes a (partial) order on the states of a layer according to their "promisingness". As featured per the documentation in listing 4.3, being "greater" according to this heuristic order is interpreted as being more promising; more likely to lead to the optimal solution.

Besides the state ranking heuristics, listing 4.3 also gives the definition of the `WidthHeuristic`. As suggested by the trait name, a `WidthHeuristic` encapsulates the heuristic which is used to determine the maximum width (W) which will be used to compile a decision diagram rooted in some state. Contrary to `Problem`, `Relaxation` and `StateRanking`, a user willing to solve a problem with *ddo* might – but is not required to – implement a custom width heuristic that works well for the problem at hand. However, chances are that in most cases, the *ddo* provided implementations (`Fixed`, `NbUnassigned`, ...) will suffice.

```
/// A state ranking is a heuristic that imposes a partial order on states.
/// This order is used by the framework as a means to discriminate the most
/// promising nodes from the least promising ones when restricting or relaxing
/// a layer from some given DD.
pub trait StateRanking {
    /// As is the case for ‘Problem‘ and ‘Relaxation‘, a ‘StateRanking‘ must
    /// tell the kind of states it is able to operate on.
    type State;

    /// This method compares two states and determines which is the most
    /// desirable to keep. In this ordering, greater means better and hence
    /// more likely to be kept
    fn compare(&self, a: &Self::State, b: &Self::State) -> Ordering;
}

/// This trait encapsulates the behavior of the heuristic that determines
/// the maximum permitted width of a decision diagram.
///
/// # Technical Note:
/// Just like ‘Problem‘, ‘Relaxation‘ and ‘StateRanking‘, the ‘WidthHeuristic‘
/// trait is generic over ‘State‘s. However, rather than using the same
/// 'assciated-type' mechanism that was used for the former three types,
/// ‘WidthHeuristic‘ uses a parameter type for this purpose (the type parameter
/// approach might feel more familiar to Java or C++ programmers than the
/// associated-type).
///
/// This choice was motivated by two factors:
/// 1. The ‘Problem‘, ‘Relaxation‘ and ‘StateRanking‘ are intrinsically tied
///    to *one type* of state. And thus, the ‘State‘ is really a part of the
///    problem/relaxation itself. Therefore, it would not make sense to define
///    a generic problem implementation which would be applicable to all kind
```

```
32 ///    of states. Instead, an implementation of the ‘Problem‘ trait is the
33 ///    concrete implementation of a DP model (same argument holds for the other
34 ///    traits).
35 ///
36 /// 2. On the other hand, it does make sense to define a ‘WidthHeuristic‘
37 ///    implementation which is applicable regardless of the state of the problem
38 ///    which is currently being solved. For instance, the ddo framework offers
39 ///    the ‘Fixed‘ and ‘NbUnassigned‘ width heuristics which are independent of
40 ///    the problem. The ‘Fixed‘ width heuristic imposes that the maximum layer
41 ///    width be constant across all compiled DDs whereas ‘NbUnassigned‘ lets
42 ///    the maximum width vary depending on the number of problem variables
43 ///    which have already been decided upon.
44 pub trait WidthHeuristic<State> {
45     /// Estimates a good maximum width for an MDD rooted in the given state
46     fn max_width(&self, state: &State) -> usize;
47 }
```

Listing 4.3: Additional traits that might be user-implemented

Finally, the *ddo* library defines traits for the `Solver` and `Frontier`. It readily provides multiple implementations of these traits and it is unlikely that a user would need or want to create a new one – which is possible however. The `Solver` trait abstracts away the details of an objects implementing algorithm 5. The library provides two implementations of this trait: `SequentialSolver` which implements the branch-and-bound with MDD paradigm on a single thread of execution. The second implementation which is provided by the framework is called `ParallelSolver`. It implements the same algorithm as `SequentialSolver`, but it is able to spread its work on multiple threads so as to use all of the available hardware.

There are also two available implementations of the `Frontier` trait. The `SimpleFrontier` which is essentially a thin wrapper around a priority queue implemented with a binary heap. The second implementation is called `NoDupFrontier`. It in addition to implementing a priority queue, that second implementation provides a mechanism that prevents the occurrence of duplicate nodes in the frontier. This check has no impact on the time or space complexity of the priority queue: the implementation uses a hash table and the duplicate test is completed in amortized $O(1)$. It might however yield significant solver speedups since it has the potential to reduce thrashing[7] exponentially.

```
1  /// This is the solver abstraction. It is implemented by a structure that
2  /// implements the branch-and-bound with MDD paradigm (or possibly another
3  /// optimization algorithm -- currently only branch-and-bound with DD) to
4  /// find the best possible solution to a given problem.
5  pub trait Solver {
6      /// This method orders the solver to search for the optimal solution among
7      /// all possibilities.
8      fn maximize(&mut self);
9      /// This method returns the value of the objective function for the best
10     /// solution that has been found. It returns ‘None‘ when no solution exists
```

[7] Fundamentally thrashing is the undesired behavior by which an exact solver repeatedly explores the same portion(s) of a problem state space.

```
11     /// to the problem.
12     fn best_value(&self) -> Option<isize>;
13     /// This method returns the best solution to the optimization problem.
14     /// That is, it returns the vector of decision which maximizes the value
15     /// of the objective function (sum of transition costs + initial value).
16     /// It returns 'None' when the problem admits no feasible solution.
17     fn best_solution(&self) -> Option<Vec<Decision>>;
18 }
19
20 /// This trait abstracts away the implementation details of the solver frontier
21 /// (a.k.a. solver fringe). That is, a Frontier represents the global priority
22 /// queue which stores all the nodes remaining to explore.
23 pub trait Frontier {
24     type State;
25
26     /// This is how you push a node onto the frontier.
27     fn push(&mut self, node: SubProblem<Self::State>);
28     /// This method yields the most promising node from the frontier.
29     /// # Note:
30     /// The solvers rely on the assumption that a frontier will pop nodes in
31     /// descending upper bound order. Hence, it is a requirement for any fringe
32     /// implementation to enforce that requirement.
33     fn pop(&mut self) -> Option<SubProblem<Self::State>>;
34     /// This method clears the frontier: it removes all nodes from the queue.
35     fn clear(&mut self);
36     /// Yields the length of the queue.
37     fn len(&self) -> usize;
38     /// Returns true iff the finge is empty (len == 0)
39     fn is_empty(&self) -> bool {
40         self.len() == 0
41     }
42 }
```

Listing 4.4: Library provided API

4.2.2 A Complete Usage Example: Knapsack

This section illustrates the use of *ddo* through a minimalistic yet extensive example showing how to model and solve the binary knapsack problem with our library. From Listing 4.5, one can observe how the traits presented in section 4.2.1 are implemented and how closely the *ddo* model matches with the mathematical abstractions from Chapter 2. In particular, it shows that the implementation of the `Problem` trait by `Knapsack` describes the DP formulation of a binary knapsack problem where a state is a plain structure with two fields: one that keeps track of the residual capacity of the sack and the other that keeps track of the number of items which have already been decided upon. This, along with the `nb_variables()` method (lines 16–18) characterizes the solution space of the problem. Similarly, the other four elements constitutive of a DP model (initial state, initial value, transition function and transition cost function) are all implemented by their eponymous methods (lines 19–33).

The remaining two methods implemented by `Knapsack` are also closely related to the mathematical definition of a DP model. The `next_variable()`

method (lines 35–39) implements the variable heuristic which is used while developing a DD. When making its decision, this heuristic is provided with an iterator over the nodes from the next (undeveloped) layer. As a consequence of this choice, our framework does not impose that a variable ordering must be fixed *a priori*. Using our implementation opens the door to defining dynamic heuristic where the variable ordering might differ across two successive relaxed DD. Finally, the `for_each_in_domain()` method is of utmost importance. It is the place where the problem explicits the possible decisions that can be made given a state and a variable[8].

```
1  #[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
2  struct KnapsackState {
3     depth: usize,
4     capacity: usize
5  }
6
7  struct Knapsack {
8     capacity: usize,
9     profit: Vec<usize>,
10    weight: Vec<usize>,
11 }
12
13 impl Problem for Knapsack {
14    type State = KnapsackState;
15
16    fn nb_variables(&self) -> usize {
17       self.profit.len()
18    }
19    fn initial_state(&self) -> Self::State {
20       KnapsackState{ depth: 0, capacity: self.capacity }
21    }
22    fn initial_value(&self) -> isize {
23       0
24    }
25    fn transition(&self, state: &Self::State, dec: Decision) -> Self::State {
26       let mut ret = state.clone();
27       ret.depth += 1;
28       if dec.value == 1 { ret.capacity -= self.weight[dec.var.id()] }
29       ret
30    }
31    fn transition_cost(&self, state: &Self::State, dec: Decision) -> isize {
32       self.profit[dec.var.id()] as isize * dec.value
33    }
34
35    fn next_variable(&self, next_layer: &mut dyn Iterator<Item = &Self::State>)
36    -> Option<Variable> {
37       let n = self.nb_variables();
38       next_layer.filter(|s| s.depth < n).next().map(|s| Variable(s.depth))
39    }
40    fn for_each_in_domain<F>(&self, var: Variable, state: &Self::State, mut f: F)
41    where
42       F: FnMut(Decision),
```

[8]This API was designed as an inversion of control in *ddo* in order to maximize the performance of the resulting solver while imposing no implementation constraint on the user.

```
43      {
44          if state.capacity >= self.weight[var.id()] {
45              f(Decision { var, value: 1 });
46              f(Decision { var, value: 0 });
47          } else {
48              f(Decision { var, value: 0 });
49          }
50      }
51  }
52
53  struct KPRelax;
54  impl Relaxation for KPRelax {
55      type State = KnapsackState;
56
57      fn merge(&self, states: &mut dyn Iterator<Item = &Self::State>) -> Self::State {
58          states.max_by_key(|node| node.capacity).copied().unwrap()
59      }
60
61      fn relax(&self, source: &Self::State, dest: &Self::State, merged: &Self::State, decision: Decision,
            cost: isize) -> isize {
62          cost
63      }
64  }
65
66  struct KPranking;
67  impl StateRanking for KPranking {
68      type State = KnapsackState;
69
70      fn compare(&self, a: &Self::State, b: &Self::State) -> std::cmp::Ordering {
71          a.capacity.cmp(&b.capacity)
72      }
73  }
74
75  fn main() {
76      let problem = Knapsack {
77          capacity: 50,
78          profit: vec![60, 100, 120],
79          weight: vec![10, 20, 30],
80      };
81      let relaxation = KPRelax;
82      let heuristic = KPranking;
83      let width = Fixed(100);
84      let mut frontier = SimpleFrontier::new(&heuristic);
85
86      let mut solver = ParallelSolver::new(&problem, &relaxation, &heuristic, &width, &mut frontier);
87      solver.maximize();
88      println!("best value {}", solver.best_value().unwrap())
89  }
```

Listing 4.5: Detailed example

The lines 53–64 of Listing 4.5 define the `KPRelax` structure and its implementation of the `Relaxation` trait. Again, the API closely matches the mathematical definitions of the $\oplus$ and Γ operators. The `merge()` method shows what it takes to provide an actual implementation of $\oplus$ and merge several nodes to derive a new relaxed node standing for them all (lines 57–59). In our example, the relaxed state is simply chosen to be the one having the

maximum residual capacity. The arc relaxation (Γ operator) is implemented by `relax()`. In the context of the Knapsack implementation, this method leaves the weight of the arcs entering the merged node unchanged.

After the relaxation operators, our example code defines the `KPranking` structure which implements the node selection heuristic from Algorithm 4. In *ddo*, this heuristic takes the form of a ranking on the nodes where a node that is *greater* comparatively to others is more likely to be kept untouched. The *lesser* nodes, on the other hand, are more likely to participate in a merge operation. Finally, the last fragment (lines 75–89) of Listing 4.5 show what it takes to instantiate the solver and use it to solve a knapsack problem instance with *ddo* using all the hardware threads available on the machine.

4.3 Engineering the MDD data structure

The astute reader will have noticed that no mention is ever made in sections 4.2.1 and 4.2.2 of an abstraction or implementation for decision diagrams. This is somewhat unexpected as DD are bound to be a key element in the implementation of a generic solver framework implementing the branch-and-bound with DD approach. And indeed, decision diagrams are central to the implementation of our *ddo* library. Their presentation has been delayed up until this point because these abstractions are not *user facing* abstractions, meaning that DDs are used under the hood by the various `Solver` implementations. And, in spite of the possibility to configure the actual DD implementation in use when solving a problem instance, it is not expected that a user would want to configure the solver and plug its own DD representation. The second reason why the presentation of the DD abstraction has beed deferred until now is that this section intends to discuss the implementation choices which can be made when implementing a structure to represent a decision diagram. Indeed, in order to implement a fast and generic solver framework, one needs to use a highly efficient DD representation. However, even well defined mathematical objects such as DDs can be implemented in many different ways. And each of the possible representation comes with its own set of strengths and weaknesses. This section uses a bit of systems programming knowledge and discusses how subtle implementation choices can affect the overall performance of a solver. In particular, it discusses four approaches that have been implemented in our library: two of which are "deep" MDD representations – namely a Naive implementation and a Vector Based one. The other two implement a "shallow" representation of the MDD. These are respectively referred to as Pooled MDD and Flat MDD.

4.3.1 The DecisionDiagram abstraction

The definition of the trait used to abstract away the implementation details of decision diagrams is given in listing 4.6. While the exact code specification of this trait might seem a little cumbersome to someone not used to Rust generics, the specification of this trait is actually quite simple. All it really does is to expose a method to *compile* an (approximate) decision diagram for some subproblem, a method to test if the compilation resulted in an exact DD and two methods to extract the value of the longest r-t path and the decisions labeling it. In addition to these, the trait also imposes the definition of a `drain_cutset()` method to iterate over the subproblems constitutive of the exact cutset of a relaxed DD – which must then be enqueued on the solver fringe.

```
1  /// This trait describes the operations that can be expected from an abstract
2  /// decision diagram regardless of the way it is implemented.
3  pub trait DecisionDiagram {
4      /// This associated type corresponds to the 'State' type of the problems
5      /// that can be solved when using this DD.
6      type State;
7
8      /// This method provokes the compilation of the DD based on the given
9      /// compilation input (compilation type, and root subproblem)
10     fn compile<P, R, O>(&mut self, input: &CompilationInput<P, R, O>)
11     where
12         P: Problem<State = Self::State>,
13         R: Relaxation<State = P::State>,
14         O: StateRanking<State = P::State>;
15     /// Returns true iff the DD which has been compiled is an exact DD.
16     fn is_exact(&self) -> bool;
17     /// Returns the optimal value of the objective function or None when no
18     /// feasible solution has been identified (no r–t path) either because
19     /// the subproblem at the root of this DD is infeasible or because restriction
20     /// has removed all feasible paths that could potentially have been found.
21     fn best_value(&self) -> Option<isize>;
22     /// Returns the best solution of this subproblem as a sequence of decision
23     /// maximizing the objective value. When no feasible solution exists in the
24     /// approximate DD, it returns the value None instead.
25     fn best_solution(&self) -> Option<Vec<Decision>>;
26     /// Iteratively applies the given function 'func' to each element of the
27     /// exact cutset that was computed during DD compilation.
28     ///
29     /// # Important:
30     /// This can only be called if the DD was compiled in relaxed mode.
31     /// All implementations of the DecisionDiagram trait are allowed to assume
32     /// this method will be called at most once per relaxed DD compilation.
33     fn drain_cutset<F>(&mut self, func: F)
34     where
35         F: FnMut(SubProblem<Self::State>);
36 }
```

Listing 4.6: The DecisionDiagram trait

In the same way the pseudo code for the compilation of a decision diagram required some inputs in algorithms 2 and 3 of Chapter 2, the `compile()`

method accepts a reference to a `CompilationInput` structure which covers the same required information. The exact definition of the `CompilationInput` structure is given in listing 4.7. As can be seen from that listing, a `CompilationInput` comprises a `CompilationType` to express that the DD must be either compiled as an exact, restricted or relaxed decision diagram. The other fields of that structure are the maximum layer width of the DD (W in the pseudo code), a reference to the DP model which is used to compile the DD (`problem` standing for input $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$ in the pseudocode), a reference to the relaxation (`relaxation` which encapsulates the formal operators $\oplus$ and Γ) and a state ranking heuristic which is used to select the nodes that are suppressed during a restriction or merged during a relaxation. Finally, the `CompilationInput` also comprises a `residual` subproblem which must be solved through the compilation of the DD.

```rust
/// This structure encapsulates the various parameters that are required when
/// compiling a MDD.
pub struct CompilationInput<'a, P, R, O>
where
    P: Problem,
    R: Relaxation<State = P::State>,
    O: StateRanking<State = P::State>,
{
    /// Tells whether the compiled DD must be an exact, restricted or relaxed DD
    pub comp_type: CompilationType,
    /// The maximum width allowed for one layer of the compiled DD
    pub max_width: usize,
    /// A reference to the DP model of the problem being solved
    pub problem: &'a P,
    /// The relaxation operators that are used when compiling a relaxed DD
    pub relaxation: &'a R,
    /// The state ranking heuristic used to select the nodes that are deleted/merged
    /// during a restrict/relax operation
    pub ranking: &'a O,
    /// The residual supproblem that must be compiled into a MDD
    pub residual: SubProblem<P::State>,
    /// The value of the best solution found so far. Used when implementing
    /// Rough Upper Bound pruning discussed later in this chapter.
    pub best_lb: isize,
}

/// An enumeration type to determine the kind of MDD that must be compiled:
/// either an exact DD, relaxed DD or restricted DD.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum CompilationType {
    Exact,
    Relaxed,
    Restricted,
}

/// This structure represents a subproblem that must still be solved. In essence
/// it is nothing but a node (represented in an MDD-implementation-agnostic way)
/// along with the best path leading to that node.
#[derive(Debug, Clone)]
pub struct SubProblem<T> {
  /// The root-state of this subproblem (Arc<T> != Edge, it is a smart pointer to a value of type T)
    pub state: Arc<T>,
    /// The value of the objective function when arriving to this current state
    /// along the best path towards this state.
    pub value: isize,
    /// The best path known to lead to this state.
    pub path: Vec<Decision>,
    /// An upper bound on the optimal value of this subproblem.
    /// (Used when implementing Rough Upper Bound Pruning and Local Bound Pruning
    /// discussed later in this chapter).
    pub ub: isize,
}
```

Listing 4.7: Inputs to the compilation of a DecisionDiagram

4.3.2 Discussion on the Implementation Details of a DD

Now that the `DecisionDiagram` abstraction of the *ddo* library has been presented, this section can discuss ways in which that trait can be implemented in practice. Concretely, the coming paragraphs will cover four different approaches to implementing a `DecisionDiagram`. The excerpts of code which are provided for each of these candidate implementations are – obviously – not extensive. They should however be sufficient to give the reader a clear idea of what is kept in memory, how the various parts interact to implement the `DecisionDiagram` trait and discuss the respective strengths and weaknesses of each implementation.

4.3.2.1 Deep MDD

A naive DD implementation consists of a network of dynamically allocated nodes each maintaining an explicit list of adjacent labeled edges forming a large chained data structure maintaining the complete graph of the MDD in memory. With this approach, each node, edge and state is an object dynamically allocated on the heap. Listing 4.8 shows an excerpt of how such an implementation (in Rust) is organized in practice. With that approach, a node is a structure holding a shared pointer to its corresponding problem state, along with the list of its inbound edges. In addition to that, a node also contains a ‘value’ field standing for the length of the longest path between the problem root and the node. It also remembers the edge which led to this node along the longest path. Likewise, an edge from the MDD is materialized by an `Edge` structure which holds a reference to its source node along with the decision labels of the edge and its associated transition cost. The structure representing the MDD as a graph is fairly straightforward: it stores a list of layers where each layer is implemented as a hash table so as to enforce the uniqueness constraint of nodes belonging to a same layer. Next to the layers, the decision diagram also keeps track of its last exact layer, and a best terminal node (if one such node exists).

```
1  /// Each node is a linked structure holding a shared (immutable) reference to
2  /// both its state and adjacent edges.
3  #[derive(Debug, Clone)]
4  struct Node<T> {
5      /// A shared referenced–counted pointer to the node’s sate
6      state: Rc<T>,
7      /// The value of the longest path between the root and this node.
8      /// In the case of the root node, this will correspond to the problem’s
9      /// initial value
10     value: isize,
11     /// An immutable shared reference–counted pointer to the best inbound edge
12     /// of this node.
13     best: Option<Rc<Edge<T>>>,
14     /// The list of inbound edges of this node
15     inbound : Vec<Rc<Edge<T>>>,
```

```
16 }
17 /// An edge is the materialization of an edge in the decision diagram. It acts
18 /// as a proxy to the parent node (from) to which it holds a
19 /// (shared reference-counted) pointer.
20 #[derive(Debug, Clone)]
21 struct Edge<T> {
22     /// A shared pointer to the source node of this edge. The destination node
23     /// is the one holding a reference to this particular edge in its 'inbound'
24     /// field
25     from: Rc<Node<T>>,
26     /// A decision is the materialization of an assignment of a value to a given
27     /// variable.
28     decision: Decision,
29     /// The cost of taking this edge on the global objective value.
30     /// It corresponds to the transition cost of the math model.
31     cost: isize,
32 }
33 /// Each layer is materialized as a hash table mapping (shared) reference to
34 /// problem states to (references to) the nodes corresponding to that state.
35 /// This is done to ensure the unicity of a state per layer.
36 type Layer<T> = FxHashMap<Rc<T>, Rc<Node<T>>>;
37 /// The decision diagram in iself is quite simple: it keeps track of all its
38 /// layers and remembers the best terminal node and a last exact layer cutset
39 #[derive(Debug, Clone)]
40 pub struct Naive<T> where T: Eq + PartialEq + Hash + Clone {
41     /// The list of layers composing this decision diagram
42     layers: Vec<Layer<T>>,
43     /// The last exact layer of the decision diagram
44     lel: Option<FxHashMap<Rc<T>, Node<T>>>,
45     /// A reference to the best terminal node of the diagram (None when the
46     /// problem compiled into this dd is infeasible)
47     best_n: Option<Rc<Node<T>>>,
48     /// Keeps track of the decisions that have been taken to reach the root
49     /// of this DD, starting from the problem root.
50     root_pa: Vec<Decision>,
51 }
```

Listing 4.8: Naive MDD implementation

Strenghts and Weaknesses While this architecture is intuitive, easy to implement and can serve as a basis to implement techniques like iterative refinement (construction by separation) and longest path trimming (LPT) [BC16b], it is far from efficient. The many dynamic allocations of small memory chunks are likely to cause memory fragmentation, impose numerous round trips to the operating system and offer poor cache locality. All of which tend to degrade the overall performance and make the accessing and freeing of any of the entities more expensive.

4.3.2.2 Variant: Vector-based architecture

To remedy this problem, one can adopt a vector-based architecture. As shown per listing 4.9, this design proceeds as follows: the MDD contains a vector of nodes which are all uniquely identified by their position in the vector. (The identifier is nothing but a strongly typed unsigned integer). Similarly, the

MDD holds a vector of edges, each of which are likewise identified by their position in the vector.

```
/// The identifier of a node: it indicates the position of the referenced node
/// in the 'nodes' vector of the 'VectorBased' structure.
#[derive(Debug, Clone, Copy, Eq, PartialEq, Hash)]
struct NodeId(usize);
/// The identifier of an edge: it indicates the position of the referenced edge
/// in the 'edges' vector of the 'VectorBased' structure.
#[derive(Debug, Clone, Copy)]
struct EdgeId(usize);
/// Represents an effective node from the decision diagram
#[derive(Debug, Clone, Copy)]
struct Node {
    /// The length of the longest path between the problem root and this
    /// specific node
    value: isize,
    /// The identifier of the last edge on the longest path between the problem
    /// root and this node if it exists.
    best: Option<EdgeId>,
    /// The identifier of the latest edge having been added to the adjacency
    /// list of this node. (Edges, by themselves form a kind of linked structure)
    inbound: Option<EdgeId>,
}
/// Materializes one edge a.k.a arc from the decision diagram. It logically
/// connects two nodes and annotates the link with a decision and a cost.
#[derive(Debug, Clone, Copy)]
struct Edge {
    /// The identifier of the node at the **source** of this edge.
    /// The destination end of this arc is not mentioned explicitly since it
    /// is simply the node having this edge in its inbound edges list.
    from: NodeId,
    /// This is the decision label associated to this edge. It gives the
    /// information "what variable" is assigned to "what value".
    decision: Decision,
    /// This is the transition cost of making this decision from the state
    /// associated with the source node of this edge.
    cost: isize,
    /// This is a peculiarity of this design: a node does not maintain a
    /// explicit adjacency list (only an optional edge id). The rest of the
    /// list is then encoded as a kind of 'linked' list: each edge knows
    /// the identifier of the next edge in the adjacency list (if there is
    /// one such edge).
    next: Option<EdgeId>,
}

/// The decision diagram in itself. This structure essentially keeps track
/// of the nodes composing the diagam as well as the edges connecting these
/// nodes in two vectors (enabling preallocation and good cache locality).
/// In addition to that, it also keeps track of the path (root_pa) from the
/// problem root to the root of this decision diagram (explores a sub problem).
/// The prev_l comprises information about the nodes that are currently being
/// expanded, next_l stores the information about the nodes from the next layer
/// and lel simply stores a last exact layer cutset.
#[derive(Debug, Clone)]
pub struct VectorBased<T> where T: Eq + PartialEq + Hash + Clone {
    /// All the nodes composing this decision diagram. The vector comprises
    /// nodes from all layers in the DD. A nice property is that all nodes
    /// belonging to one same layer form  a sequence in the `nodes` vector.
    nodes : Vec<Node>,
```

```
58     /// This vector stores the information about all edges connecting the nodes
59     /// of the decision diagram.
60     edges : Vec<Edge>,
61     /// Keeps track of the decisions that have been taken to reach the root
62     /// of this DD, starting from the problem root.
63     root_pa: Vec<Decision>,
64     /// Maintains the association nodeid−>state for the nodes of the layer which
65     /// is currently being expanded. This association is only used during the
66     /// unrolling of transition relation, and when merging nodes of a relaxed DD.
67     prev_l: FxHashMap<NodeId, T>,
68     /// The nodes from the next layer; those are the result of an application
69     /// of the transition function to a node in ‘prev_l‘.
70     /// Note: next_l in itself is indexed on the state associated with nodes.
71     /// The rationale being that two transitions to the same state in the same
72     /// layer should lead to the same node. This indexation helps ensuring
73     /// the uniqueness constraint in amortized O(1).
74     next_l: FxHashMap<T, NodeId>,
75     /// The last exact layer of the decision diagram
76     lel: Option<Vec<(T, NodeId)>>,
77     /// The identifier of the best terminal node of the diagram (None when the
78     /// problem compiled into this dd is infeasible)
79     best_n: Option<NodeId>,
80 }
```

Listing 4.9: Vector-Based MDD implementation

In this design, nodes do not hold shared references to their edges, but use their identifier instead. And similarly, the edges do not store an actual reference towards the nodes they refer to; but instead store their unique identifiers only[9]. Moreover, instead of letting the nodes actively maintain an adjacent edges list, the edges behave as nodes of a singly linked list. Each edge knows of a potential next edge in the adjacency list of its destination node. (same could be done at parent level). Hence, a node only keeps the identifiers of *two* (potentially the same) edges: one identifier corresponds to the head of the adjacency list, and the other is used to remember the edge which must be taken to reconstruct the best path between the node and the root of the MDD. The rationale behind this apparently odd construct is to let the nodes and edges be small, fixed size structures which are easily copied by value and can easily be moved while maintaining their semantic integrity (no dangling pointer after move[10]). All that while maintaining the capability for a node to list *all* its inbound edges.

In addition to the list of nodes and edges, this architecture also uses an hash table during the top down compilation of the DD to ensure its reduction. When nodes from layer L_i are expanded, the nodes from layer L_{i+1} are all added to the hash table indexed on the states. This way, whenever the MDD produces a state which is reachable trough more than one path, it is

[9] Getting rid of the shared references ‘shared_ptr‘ or ‘Rc‘ is helpful since it avoids the individual heap allocation for each of the referenced elements and easily maintain their semantics even after a move of the DD

[10] Between two threads for instance

guaranteed that all paths end up in the exact same node of the diagram.

It is also interesting to note that a `Node` does not store any reference to the state associated with the node. This is actually not necessary with this design. Indeed, the states only ever exist in the `prev_l`, `next_l` and `lel` fields of the `VectorBased` DD which covers all the use cases when states are needed. Indeed, the node associated with a state is only accessed when:

1. computing a transition/transition cost based on the problem definition.
2. ensuring the uniqueness of a given state among the nodes of a layer.
3. selecting and merging nodes of a relaxed MDD.
4. selecting nodes to delete from the next layer in a restricted MDD.

This approach offers several advantages: as soon as the state has become useless and will not be used anymore, the associated memory can be freed. This helps mitigating the preallocation impact in practice since user-defined state typically accounts for a large fraction of the total allocated memory. Moreover, given that the association $state \longrightarrow node$ only exists within *one layer at the time*; the information can simply be *moved* from one place to the other. It is therefore not necessary to store/access them through reference counted smart pointers. Which again reduces the number of dynamic allocations and improve the cache locality of the MDD compilation.

Strengths The vector based architecture is particularly effective as it avoids the weaknesses discussed about the `Naive` implementation. Vectors are very simple data structures and they typically offer good cache locality. Moreover, any reasonable vector implementation preallocates memory to avoid round trips to the operating system while accomodating further insertion in the vector. Given that DD compilation requires lots of such insertions, the benefits are not negligible. Furthermore, even though the vector-based architecture uses a hash table to reconcile equivalent states; that table only ever contains one layer at a time. Which means the hash table is going to remain fairly small (depending on the maximum layer width W). Hence, the likelihood of the hash table requiring to be resized (and consequently rehashed) is largely decreased. This again contributes to giving an efficiency gain to the vector based architecture.

4.3.2.3 Pooled-MDD

As explained in [Ber+14b], any node u_i of a DD having one single outgoing arc $a = (u_i, u_j)$ can be removed from the graph and replaced by a longer arc provided that $\sigma(u_i) = \sigma(u_j)$ and $v(a) = 0$. In other words, a node can be

removed from a DD and replaced by a long arc spanning over several layers if it has one single transition which does not alter the node state and involves no cost (as understood by the objective function).

Based on this observation, one can devise a pooled MDD implementation. A pooled MDD is one that does not necessarily expand all the nodes of a given layer. Instead, the expansion of the nodes for which a long arc can be inserted is deferred for as long as possible. The rationale behind that choice being to avoid "bloating" layers with nodes that add no information. Indeed bloated layers would quickly exceed the maximum width limit, and would consequently need to be either restricted or relaxed; causing an information loss. By avoiding to bloat the layers in the first place, one delays the time when restriction/relaxation is needed.

As shown in Algorithm 9, the compilation of a pooled MDD closely resembles the procedure detailed in Chapter 2. There are however differences to be noted. First, the implementation maintains two sets of nodes: the first set represents the current layer, the set of nodes that cannot be removed and replaced by a longer arc. The second set of nodes serves as a *pool* of nodes that belong to some *subsequent* layer. The `select_nodes_from()` function returns the subset of nodes from the pool that will be impacted by a decision on variable x_i (either in terms of the state or the objective value) and must be moved to the current layer. Also, because the number of such nodes is not known beforehand, and because it might exceed the maximum layer width, a call to the width bounding procedure – either `restrict()` or `relax()`, which is here generically referred to as `squash_layer()` in Algorithm 9 – must be inserted before proceeding to the development of the nodes.

To understand how long arcs are created in Pooled-MDD, one needs to grasp the dynamics of Algorithm 9 and observe that when nodes are *created* (line 13), they are not inserted in the next layer but moved to the pool instead. Also, before making any decision about a variable x_i, the nodes that might be impacted by a decision on x_i are drawn from the pool to form layer L_i (line 6). The subtlety to understanding how long arcs are introduced resides in realizing that the nodes drawn from the pool to form layer L_i were not necessarily inserted in the pool during the expansion of layer L_{i-1}. For instance, if the node u is inserted in the pool because it was created after a decision on variable x_5, it may very well be the case that u remains in the pool until a decision is to be made about x_{19}. In that case, there would exist a long arc between the parents of u at layer L_5 spanning until layer L_{19}.

Strengths As explained above, the selection of nodes from the pool might defer the development of some nodes that would otherwise have belonged to the current layer. Doing so, a pooled MDD implementation is able to avoid bloating layers and hence to maintain a higher degree of diversity among the

solutions encoded in a restricted or relaxed MDD[11]. Moreover, this implementation is generally quite memory efficient in the sense that it only needs to maintain one layer in addition to the pool of future nodes.

Weaknesses Although the width of the layers is bounded by a value W, the size of the pool is not bounded by anything. This might be problematic as the size of that pool might grow too fast to be tractable. Third, because the only materialized layer of a pooled-MDD is not always complete (by design, because of the long arcs), it is much more expensive to produce a LEL cutset using a pooled-MDD than any other kind of implementation (this does not apply to FC which can be computed on-the-fly). Finally, because the efficiency of the `select_nodes_from()` procedure relies on its ability to swiftly discriminate the nodes that must remain in the pool from others, it is necessary that the user of a solver backed by pooled-MDDs provides an implementation of that check. This is a requirement which is not enforced by alternate-MDD implementations.

Algorithm 9 Construction of a Pooled-MDD

1: **Input:** a DP model $\mathcal{P} = \langle S, r, t, \bot, v, \tau, h \rangle$
2: **Input:** a maximum layer width W
3: **Input:** relaxation operators $(\oplus, \Gamma)$ when compiling a relaxed MDD
4: $Pool \leftarrow \{r\}$
5: **for** $i \in \{0 \ldots n-1\}$ **do**
6: $L_i \leftarrow select_nodes_from(Pool, i)$
7: $Pool \leftarrow Pool \setminus L_i$
8: $L_i \leftarrow squash_layer(L_i, \oplus, \Gamma)$
9: **for** $u \in L_i, d \in D_i$ **do**
10: $u' \leftarrow$ a node associated with state $\tau_i(\sigma(u), d)$
11: **if** $\sigma(u') \neq \bot$ **then**
12: $U \leftarrow U \cup \{u'\}$
13: $Pool \leftarrow Pool \cup \{u'\}$
14: $a \leftarrow (u,' u, d)$
15: $v(a) \leftarrow h(\sigma(u), d)$
16: $A \leftarrow A \cup \{a\}$

4.3.2.4 Flat-MDD

Similar to the pooled approach described in 4.3.2.3, the data structure we call *flat-MDD* only maintains a fraction of the actual graph (the current and next

[11] This is to be related to the quality-measure proposed in [BC16b]

layers). However, this implementation closely respects the compilation procedure described in Chapter 2. Thus, one should really think of the construction of a flat-MDD as maintaining a slice of an equivalent naive-MDD. The correctness of this implementation relies on the observation that the Branch-and-Bound with MDD algorithm (Algorithm 5, p. 18 in Chapter 2) really only focuses on finding the optimal solution x^* of the MDDs it builds. Given all that information is gradually built in each of the layers, it is perfectly fine for an implementation to forget about layer L_{i-1} when deriving the next layer L_{i+1} based on the nodes of the current one L_i. This is the main idea that underlies the flat-MDD implementation.

In practice, the nodes of a flat-MDD implement a singly linked data structure where each node remembers the single last arc constitutive of the best path between the root and itself. This memorized piece of information is subject to potential change for the node u' whenever an arc (u, u') is added to $\mathcal{A}$. Thanks to that information, one only needs to traverse the chain of remembered arcs in order to recover the best path (hence best partial assignment) between a node and the root of the MDD.

Despite maintaining only a fraction of the information held in an equivalent naive representation, a flat MDD is perfectly able to efficiently produce an exact LEL cutset. To that end, it suffices to remember the value of the current layer instead of wiping it off when $squash_layer()$ modifies the next layer to be. In practice, this does not even require to fully copy the nodes from the current layer and can be achieved by a simple swap of two integers.

Strengths The strengths of the flat-MDD data structure are multiple. First, and as is the case with the naive implementation, a flat-MDD provides a strong guarantee on the memory required to compile a bounded width approximate MDD. Still, because it only needs to maintain a fixed size slice of the MDD, it is able to forget all the irrelevant nodes from previous layers. And because each of these nodes implements a singly chained data structure, they are able to share common partial assignment (prefixes), thereby further reducing the memory requirements associated to the MDD representation. Notwithstanding that, a flat MDD is capable of returning LEL and FC cutsets very efficiently. Finally, because of its sliced nature, a flat-MDD implementation offers many opportunities for recycling allocated objects; typically boosting the overall performance of a solver using those.

Weaknesses Similar to pooled MDD, a flat MDD has the ability to forget some portions of the DD which are guaranteed not to yield the best solution of the subproblem materialized by the MDD. Because of that ability, not all paths are present in the final structure and some information is irremediably lost. This might be problematic when implementing further pruning tech-

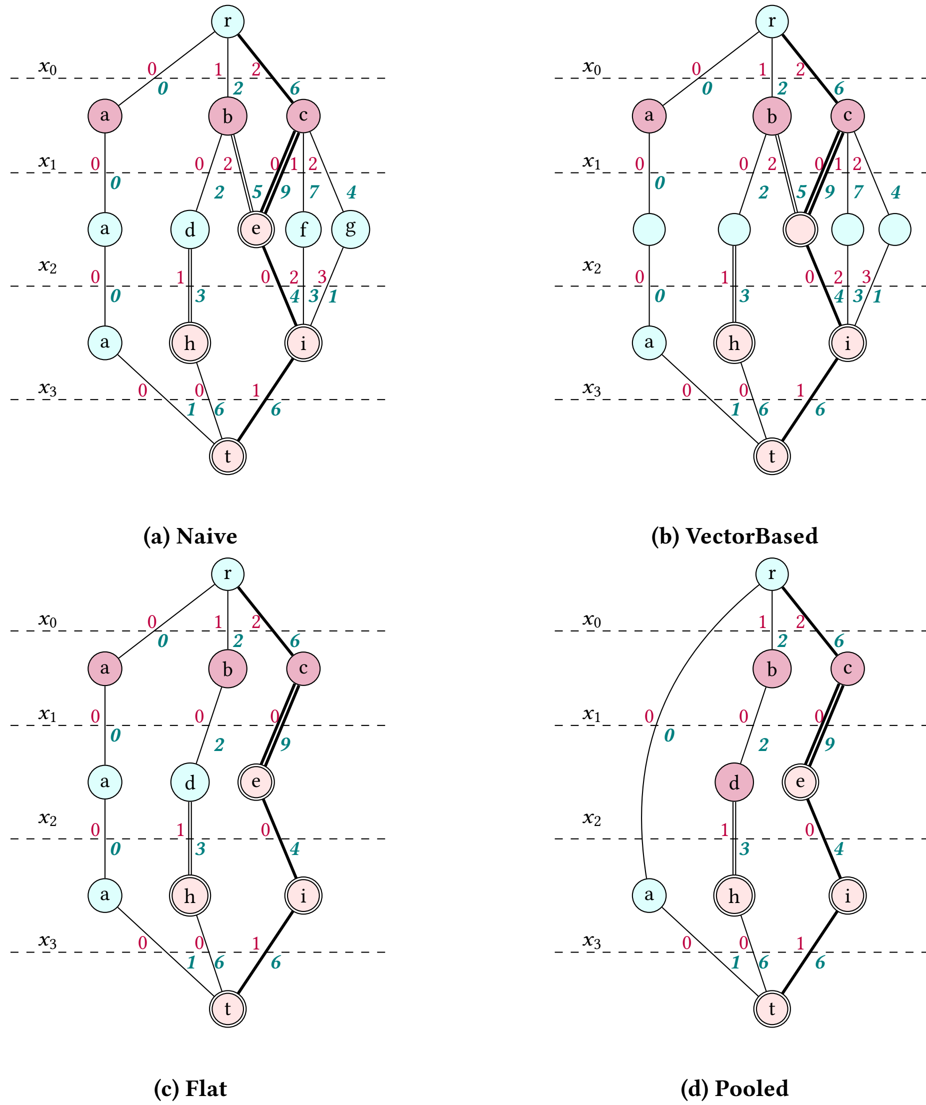
niques (e.g. local bounds, see the next chapter). The other main weakness of a flat-MDD is relative to pooled-MDD. Indeed, because flat and naive-MDDs do not encode any long arc, their merge (for relaxed-MDDs) and deletion strategies (for restricted-MDDs) are more aggressive than that of a pooled-MDD implementation. This might result in the bounds obtained from flat, naive, or vector-based MDD to be somewhat less tight than those obtained from a pooled implementation.

4.3.2.5 Summarizing Example

The figure 4.2 summarizes the peculiarities of each DD implementation that has been discussed in sections 4.3.2.1 through 4.3.2.4. These four graphs represent the same example relaxed DD when compiled with each implementation. The decision labels (red) of the arcs are shown above the layer separation lines (dashed). The arc weights (green) are shown below the separation lines. If an arc is double stroked, it means that arc enters a node which is the result of a merge operation ($\oplus$) as a consequence of which it has been Γ relaxed. Exact nodes are depicted with a single line whereas inexact nodes are represented with a double stroke. The background of the nodes belonging to the exact cutset of the DD is colored red. The letter inside of a given node denotes the fact that the *state* associated with that particular node is known. On the contrary, when a node is shown without a letter in the middle, it means that the DD implementation remembers the *structural* information about this node but not its associated state. Finally, the arcs along the longest path of the MDD are boldfaced.

A comparison of the example depicted in figure 4.2 highlights the key differences between the four implementations. The first element that stands out when comparing 4.2a and 4.2b is that both implementations encode the exact same graph. Both of these implementations maintain the complete graph in memory. There are, however, two differences between these implementations: The first one is purely technical – and thus not visible on these graphs: the naive implementation (section 4.3.2.1) allocates all nodes and edges using *dynamic (heap) memory allocation* whereas the VectorBased implementation stores them in contiguous vectors of pre-allocated memory (hence the name). The second difference between these two implementations stems from the fact that the VectorBased approach forgets the state associated with nodes it does not need anymore. As shown in figure 4.2b, the states from the nodes belonging to the exact cutset, as well as the states of the last and before last layers are maintained. All other states however (nodes from the middle layer) are irremediably lost.

The second element, which should be noted when observing these graphs, is that the *shallow* MDD representations (that is, Flat and Poooled) are not as



(a) Naive **(b) VectorBased**

(c) Flat **(d) Pooled**

Figure 4.2: Summary of the Four Proposed Implementations

shallow as one might imagine. While it is true that the DD only gives direct access to the nodes in the last two layers (+ the exact cutset), all nodes maintain a pointer to their best parent. This is necessary in order to be able to retrieve the decision along the optimal path from t. It should, however, be noted that only the best paths to the "bottom" nodes are represented with these approaches. All paths have been explored, but many of them are forgotten as soon as they are shown to not be "the best". The same goes about nodes that are not part of "the" best path towards one of the bottom nodes. For instance, this is why nodes f and g are nowhere to be seen in subfigures 4.2c and 4.2d.

There are two more key differences which differentiate the pooled approach from all others. The first one is that the pooled implementation can introduce long arcs spanning over multiple layers, while all other considered implementations do not. This is visible when comparing subfigure 4.2d with any of the others. Indeed, all implementations explicitly represent the partial assignment $[\![x_0 = 0, x_1 = 0, x_2 = 0]\!]$ and thus, introduce multiple copies of state a. On the other hand, the pooled approach chooses to introduce a long arc between r and a and only ever expand that node when taking a decision about variable x_3. Doing so is definitely valid, as it appears that decisions about x_1 and x_2 have no impact on the resulting state (any decision on these variables from state a yields the state a) nor on the objective value (the cost associated with these decisions is 0). The second consequence of the introduction of such long arcs, is that implementing a LEL cutset is much harder with a pooled implementation than in the other cases. This is because when compiling the DD with a pooled implementation, there is no guarantee that a "layer of expanded nodes" comprises all nodes that should actually have pertained to that layer. Indeed, some nodes might remain in the pool rather than participate in the development of the layer – as would e.g. occur for node a. This is why the exact cutset which is stored by a pooled implementation is a *Frontier Cutset* rather than a *Last Exact Layer Cutset*.

4.4 Alternatives to *ddo*

Ddo is the first fully generic open source framework to develop solvers based on the branch-and-bound with MDD paradigm, but it certainly is not the first implementation of the DDO paradigm. Indeed, Bergman, Cire, and Van Hoeve developed their own C++ solution to evaluate the relevance of their novel algorithm [Ber+16b]. While that solution shares some commonalities with *ddo*, there are differences as well. From a technical point of view, the MDD implementation of that solution closely resembles the *Naive* implementation sketched in Listing 4.8 (p. 59). However, that solution was not meant as a complete framework for the development of DD-based solvers.

For instance, the code to deal with the different problems they investigated shares the abstract definition of a Binary Decision Diagram (BDD) as well as some useful utility functions. Still, that common behavior is not self-sufficient and the solver of each problem must provide its own complete implementation of the `generate_exact()`, `generate_relaxed()`, and `generate_restricted()` methods. This is why most of the solvers they wrote use a BDD data structure similar to our `Naive` implementation, but their solver for the Maximum Independent Set Problem (MISP) uses a *Pooled* approach.

This state of affairs stems from the fact that the C++ solution of the original authors of the method does not provide a clear abstract specification of the problem and its relaxation. Neither does it provide a clear specification of the heuristics that can be used to tune the behavior of their solvers. It results that their solution is quite efficient but probably not as convenient to use as *ddo* since their approach requires more input than just a `Problem` definition and its `Relaxation`. A fortiori given that this solution offers no abstraction for the `Solver` (it is just the main function) or the `Fringe`.

Along similar lines, Horn et al. have used their own C++ implementation of restricted and relaxed decision diagrams to validate the effectiveness of their $A*$-based compilation scheme. The latter is to be related to the rough upper bound pruning technique presented in Chapter 5. The source code of their implementation was, however, not publicly available, which is why a detailed comparison of their solution with *ddo* cannot be done at the time being.

In [CGS22], Coppé et al. also used a custom C++ implementation to solve the Constrained Single Row Facility Layout Problem (CSRFLP) with Decision Diagrams. The search method which they use differs from the pure branch-and-bound approach implemented in *ddo*. Their approach is closer to an A^* search where restricted DDs are used to find good primal solutions. From a technical perspective, the MDD implementation of their solution is a `VectorBased` DD implementation whose implementation was based on that of *ddo*[12].

In [OH19], O'Neil and Hoffmann have applied the branch-and-bound with MDD paradigm from [Ber+16b] using *narrow-width* DDs to solve the Travelling Salesman Problem with Pickup and Deliver in real-time (TSPPD-rt). Their implementation is written in Go, and although it also leverages parallel computing, their approach slightly differs from that of *ddo* (Algorithms 6, 7, and 8 presented earlier in this chapter on pages 42, 44, and 45). Rather than having long-running threads periodically polling the content of the Fringe, the implementation of O'Neil et al. performs a form of *sharding*[13] to try

[12]The paper explicitly mentions to have drawn its inspiration from *ddo*.

[13]Decomposition of the workload into several batches that are dispatched to different work-

to balance the workload. Each shard is given to a goroutine (a.k.a. fibers, lightweight threads) which processes the complete shard and then stops. Whenever a goroutine identifies new bounds to explore (using the nodes from an exact cutset), this data is asynchronously passed on to the main thread using a channel. The consequence of that choice is that the solver must perform an additional loop where it periodically spawns bursts of worker goroutines. The benefit of that approach resides in its programming simplicity and the low synchronization overhead from using asynchronous channels. The drawback of that approach, however, is that it can cause suboptimal use of the available CPU since some shard might concentrate the vast majority of the effort to be done; thereby leaving the other cores idle.

The HADDOCK system proposed in [GMH20] shares many common aspects with *ddo*. Indeed, both libraries facilitate the task of a developer working on a solver that uses MDDs and more specifically *Relaxed MDDs*. Moreover, both libraries aim at providing a declarative – yet very different – approach to reasoning about decision diagrams. Despite their commonalities, HADDOCK and *ddo* fill different niches: while the goal of *ddo* is to facilitate the development of solvers based on the branch-and-bound with MDD paradigm, HADDOCK targets the development of MDD propagators in the context of constraint programming solvers. Because of that difference in perspective, the MDDs compiled by HADDOCK do not necessarily correspond to dynamic programs (DP) whereas all DD compiled by *ddo* are thought of as DP. From a technical point of view, HADDOCKS is written in C++ and integrated with minicpp [MSVH21]. It represents decision diagrams using large chained a data structure that resembles the *Naive MDD* described earlier in this chapter. But the resemblance ends there as the DD compilation procedure of HADDOCK is not dictated by an abstract DP. Instead, it is driven by custom MDD propagator algorithms such as those of Hoda, Hoeve, and Hooker for the *Among* constraint [HHH10] which the authors use as an example in their original paper presenting the HADDOCK system.

4.5 Experimental Evaluation

In order to evaluate the efficiency of the various MDD implementations that have been proposed in this chapter, we conducted an experimental study bearing on instances of the MISP, MCP, MAX2SAT, TSPTW and PSP as they have been detailed in Chapter 3 of this thesis.

The experimental protocol we used was the following: for each instance, we compiled one relaxed MDD with each implementation, using various maximum layer widths ($W \in \{100; 1000; 10,000; 100,000\}$) for each pair of in-

ers.

stance and implementation. For each of these compilations, we measured

- the time required to compile the MDD,
- the maximum amount of allocated RAM[14],
- the length of the longest r-t path in the MDD,
- the size of the exact cutset which could be extracted from the DD,
- and whether the compiled DD was an exact DD or not.

Each of these compilations have been carried out using one single thread of the *same physical machine* running Linux. The latter is equipped with an Intel Xeon E5540 CPU and 24G of RAM. The duration of each compilation was capped by a 180 seconds timeout. All implementations have been written in Rust using our common generic framework. This allowed to isolate the impact of the DD implementation, leaving all other factors untouched (exact same model, exact same set of heuristics, etc...).

4.5.1 Length of the r-t paths

Figure 4.3 compares the length of the longest r-t path in the decision diagrams compiled with the Naive implementation (used as a baseline) and the alternative implementations. On this whisker plot, the values on the y axis represent the ratio $\frac{\text{Length X}}{\text{Length Naive}}$. Thus, higher y values indicate cases where a longer r-t path was found with an alternative implementation versus the Naive one; which must be interpreted as the ability for that specific implementation to derive *tighter* bounds from a relaxation.

From this graph, it clearly appears that no implementation has an edge over the others in terms of tightness of the derived bound. Indeed, the first, second, and third quartiles are all centered exactly on the 1 ratio indicating that the length of the longest r-t path in the compiled DD (hence the derived bound) is the same for all implementations. The only few unaligned marks on this graph stem from a slight difference occurring at the time of selecting the subset of nodes that must be merged during relaxation. Indeed, the node-selection heuristic for the MISP defines a *partial order* and not a *total order*. Hence some implementations may have broken ties in different ways and thus compiled slightly different DDs which is why the longest r-t path is not always the same. However, these cases are exceptional and in the vast majority of times, the compiled DDs are the exact same.

[14]This was measured thanks to a custom allocator which wraps the system allocator and keeps track of the current and maximum amount of required RAM.

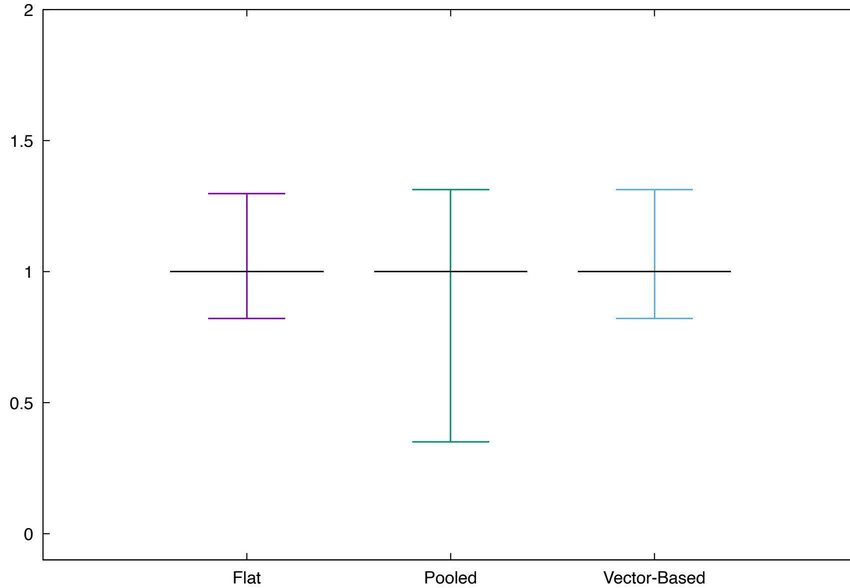


Figure 4.3: Compared length of the longest r-t paths ($\frac{\text{Length X}}{\text{Length Naive}}$)

4.5.2 Size of the Exact Cutset

Figure 4.4 compares the size of the exact cutsets extracted from the compiled DDs. Similar to the above graph, the distance on the y axis represents the ratio $\frac{\text{Cutset Size X}}{\text{Cutset Size Naive}}$. For instance, on this graph, a y value of 2 indicates that the exact cutset extracted from the alternate implementation comprises twice as many nodes as that of the Naive MDD.

The first observation to make about this graph is that *in every case*, the Naive, Flat, and VectorBased implementation create exact cutsets of the same size. This is expected since the exact portion of the MDDs compiled by these three methods corresponds to the exact same mathematical object in all three cases. The second observation to make about Figure 4.4 is that the Pooled implementation exhibits behavior strongly differing from all other implementations. On average, the size of an exact cutset extracted from the Pooled implementation is about 8 times as large as that of any other implementation. But in certain cases, the ratio amounts to 500 times the baseline. This difference in behavior was to be expected given that Pooled produces a *Frontier Cutset* (FC) whereas all other implementations create a *Last Exact Layer Cutset* (LEL). It is, however, striking to see how much larger the FC can be versus the LEL. We hypothesize that the large number of nodes that are added to the solver frontier because they pertained to a FC might be a reason why LEL were observed to fare better than FC in practice [Ber+16b].

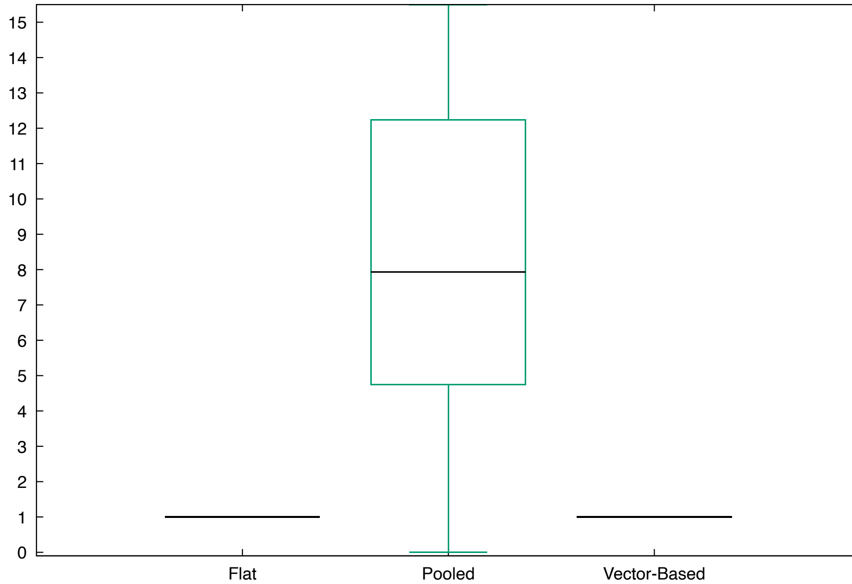


Figure 4.4: Compared size of the exact cutset ($\frac{\text{Cutset Size X}}{\text{Cutset Size Naive}}$)

4.5.3 Memory Usage

Because the compilation of an MDD is both a CPU and memory-intensive process, and in order to measure the impact of the implementation choice on the memory consumption; Figure 4.5 compares the maximum amount of memory required to compile a given MDD. Similarly to the previous graphs, the y axis represents the ratio $\frac{\text{Max RAM X in megabytes}}{\text{Max RAM Naive in megabytes}}$.

The first observation to make about Figure 4.5 is that both the Pooled and Flat implementations achieve their goal of reducing memory consumption. And in particular, the Flat implementation does it the best requiring on average no more than 22% of the memory required to compile a Naive MDD. Likewise, the Pooled implementation requires on average 46% of the baseline memory. The VectorBased case is interesting as well: even though it sometimes requires five times as much memory as the Naive implementation, the central tendency (median) indicates that most of the time, it only uses 66% of the baseline memory requirement.

To better understand the memory-allocation dynamics of the VectorBased implementation, it is interesting the plot the same information as in Figure 4.5, but in a slightly different way. On Figure 4.6, the distance along the x axis represents the amount of RAM required to compile the MDD with a Naive implementation whereas the y axis quantity represents that requirement with another implementation. A mark below the diagonal indicates

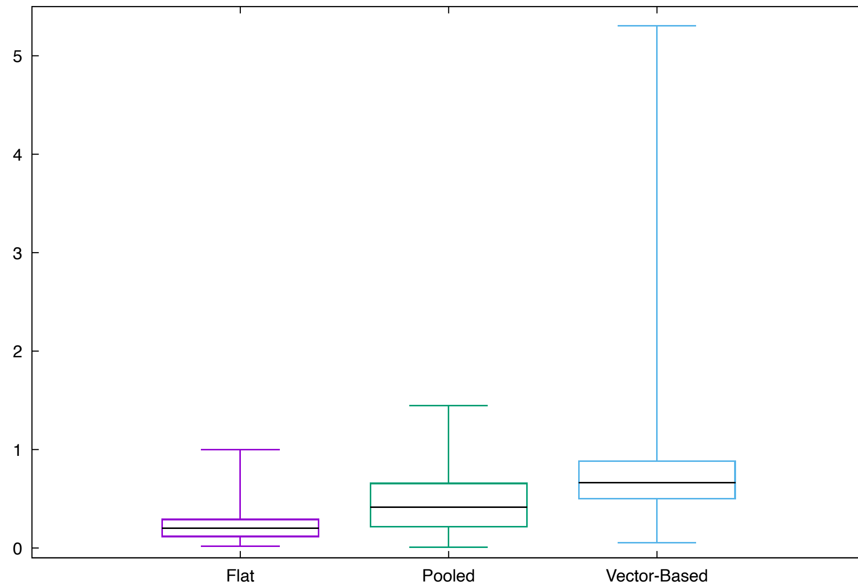


Figure 4.5: Compared maximum memory usage ($\frac{\text{Max RAM X in megabytes}}{\text{Max RAM Naive in megabytes}}$)

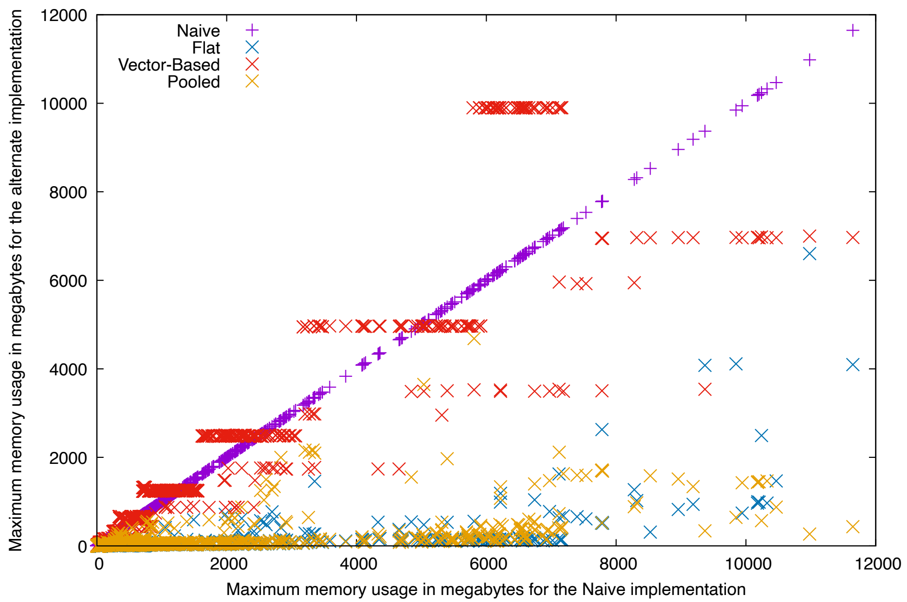


Figure 4.6: Compared maximum memory usage (in megabytes) with Naive implementation as a baseline

a lower (better) memory consumption and a mark above that line a higher memory requirement. From that figure, we can observe that the VectorBased memory usage exhibits a "staircase" pattern. This is caused by the memory preallocation happening in the vectors of the VectorBased implementation. Interestingly enough, the steps of the staircase are somehow centered on the diagonal, confirming the central tendency observed above. From Figure 4.6, one can also observe that the extreme cases where the VectorBased implementation requires five times as much memory as the baseline only occur for small values. Which is why these extreme events can hardly be seen in Figure 4.6.

4.5.4 MDD Compilation Time

Figure 4.7 compares the time to compile a given MDD with the baseline (Naive) implementation versus the other proposals. As for the previous graphs, the quantity on the y axis represents the ratio between the duration when the DD is compiled with the target implementation and the duration with the Naive implementation ($\frac{\text{Duration X}}{\text{Duration Naive}}$).

There are a few observations to make about this graph, the first one being that the VectorBased implementation is *almost always faster* than the baseline even though both implementation compute the same information. That performance improvement is significant: the central tendency shows that in most cases the VectorBased implementation takes 85% of the base line duration of less. This value drops to about 75% when the analysis is restricted to the MDD compilation that lasted for 5 seconds or more (Figure 4.8). Such performance gains are obviously desirable. Again, these are the consequence of the preallocation scheme used in the vectors used by the VectorBased implementation. A comparison between Figure 4.7 and Figure 4.8 also shows that the extreme cases where the VectorBased implementation is much slower than the Naive one only ever happen when durations are very short which makes these outliers much less interesting.

It is also worth noting that the Flat and Pooled implementation are often faster than the baseline but are usually a fraction slower than the VectorBased implementation. In the event that the compilation of a Flat MDD is slower than in the case of a Naive implementation, the overhead is usually small. That overhead can however grow quite large in the case of a Pooled implementation. We attribute this worse-than-expected behavior to the absence of checks on the size of the `pool` in the Pooled implementation. Because of that, the compilation might end up spending quite a bit of time iterating over the nodes of the pool in order to identify the ones which must be expanded in the next layer. This step is not required with any other implementation, and given the width-boundedness of the layers, the layer expansion is guar-

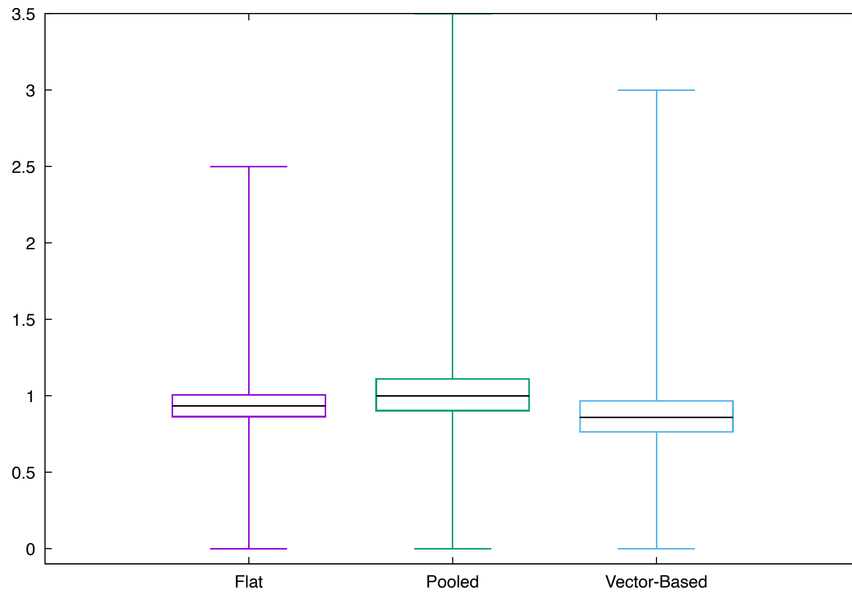


Figure 4.7: Compared compilation duration ($\frac{\text{Duration X}}{\text{Duration Naive}}$)

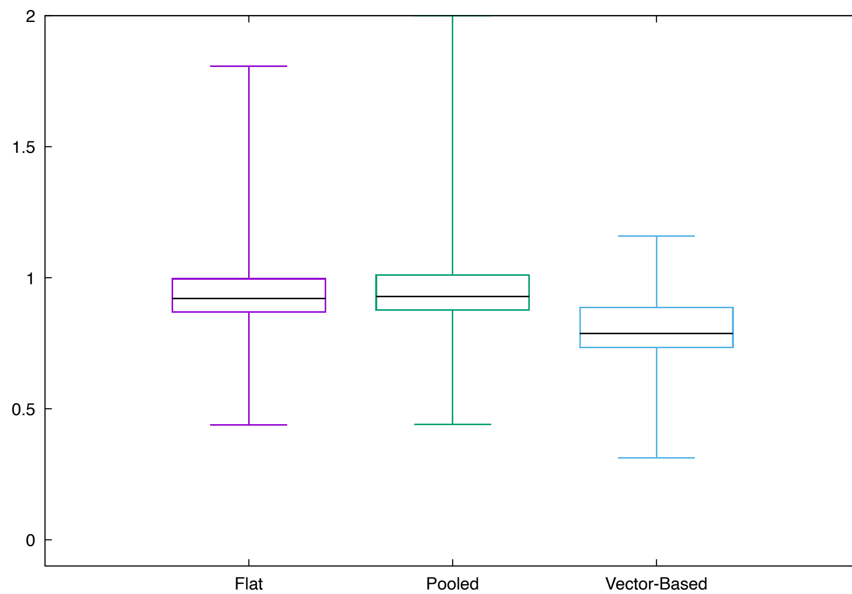


Figure 4.8: Compared compilation duration ($\frac{\text{Duration X}}{\text{Duration Naive}}$) considering only the compilation that lasted for at least 5 seconds

anteed to not take an excessive amount of time with these representations. This guarantee is not enforced on the pool of the Pooled DD which might be the reason for lots of wasted time.

Table 4.1 completes the above information by showing for each method a summary of the number of compilations that have succeeded and timed out. While the figures in this table essentially portray the same information as Figures 4.7 and Figure 4.8, they also show that the performance gains from the Flat and VectorBased implementations can have a significant impact. And indeed, the VectorBased implementation was able to compile about 300 MDDs more than the baseline without exceeding the time out.

Method	**Successful**	**Timeout**
Naive	6187	1361
Flat	6264	1284
Pooled	5460	2088
Vector-Based	**6470**	**1078**

Table 4.1: Number of successful MDD compilation and timeouts

4.6 Conclusion

This chapter started with a presentation of the *ddo* library. Then, it presented and evaluated four different implementations of the same mathematical *decision diagram* object. In particular, it presented two deep implementations (`Naive` and `VectorBased`) and two shallow representations (`Flat` and `Pooled`). This chapter also showed that the resolution of a dynamic programme through DD compilation is both a CPU and memory intensive process. Therefore, an efficient implementation must pay attention to both algorithmic and non-algorithmic costs (system calls, page faults, cache miss). Indeed, the costs incurred by the latter category might grow large and are easily overlooked.

The experimental study we conducted revealed that the `VectorBased` implementation is extremely effective. Indeed, it significantly outperforms all other implementations. This high efficiency is essentially achieved though the memory pre-allocation happening in the vectors composing the DD structure. This allows to reduce the number of round trips to the Operating System i.e. by decreasing the number of required system calls and page faults since allocations are made in large blocks. This also improves the data locality since nodes that are created close to one another are located next to one another in the vectors (continuous chunks of RAM). This proves beneficial since the aforementioned nodes are also likely to be *used* at close times and hence to

benefit from burst read[15].

Even though their time performance was beaten by that of the Vector-Based implementation, the shallow approaches really shined in terms of memory consumption. Their much lower memory usage might thus be exploited to compile larger DDs than would be done with a deep representation using the same amount of RAM. Nevertheless, one needs to be careful when selecting a `Pooled` implementation rather than a `Flat` or `VectorBased` one as the pool management and the exact cutset that are generated might dramatically impact the performance of the overall resolution process.

[15] This latter point might seem anecdotal but additional experiments where solvers are run in parallel and can be executed on different cores over the course of their execution have highlighted the impact of cache locality.

Improving the Filtering of Branch-and-Bound with MDD

5

Contributions and Publication Information

This chapter is largely based on Xavier Gillard et al. "Improving the filtering of Branch-and-Bound MDD solvers". In: *CPAIOR*. 2021. Its contribution consist of two bounding techniques: *rough upper bounds* and *local bounds* which can be used to improve the filtering of branch-and-bound MDD solvers.

This chapter presents two algorithmic methods to improve the efficiency of optimization solvers based on the branch-and-bound with MDD paradigm. These methods aim to improve the filtering capabilities of these solvers, increasing the fraction of the problem state space covered by each MDD compilation. In particular, this chapter presents and evaluates the effectiveness of the *Local Bounds* (LocB) and the *Rough Upper Bounds* (RUB).

5.1 Local bounds (LocB)

LocB is a new and effective rule that leverages the structure of bounded width MDDs to avoid the exploration of non interesting nodes. Conceptually, the technique works as follows: a relaxed MDD $\overline{\mathcal{B}}$ provides us with *one* upper bound $v^*(\overline{\mathcal{B}})$ on the optimal value of the objective function for some given sub-problem. However, in the event where $v^*(\overline{\mathcal{B}})$ is greater than the best known lower bound $\underline{v}$ (best current solution) nothing guarantees that all nodes from the exact cutset of $\overline{\mathcal{B}}$ admit a longest path to t with a length of $v^*(\overline{\mathcal{B}})$. Actually, this is quite unlikely. This is why we propose to attach a *"local" upper bound* to each node of the cutset. This local upper bound – denoted $v|_u^*$ for some cutset node u – simply records the length of the longest r-t path passing through u in the relaxed MDD $\overline{\mathcal{B}}$.

In other words, LocB allows us to refine the information provided by a relaxed DD $\overline{\mathcal{B}}$. On one hand, $\overline{\mathcal{B}}$ provides us with $v^*(\overline{\mathcal{B}})$ which is the length of the longest r-t path in $\overline{\mathcal{B}}$. As such, it provides an upper bound on the optimal

value that can be reached from the root node of $\overline{\mathcal{B}}$. With the addition of LocB, the relaxed DD provides us with an additional piece of information. For each individual node u in the exact cutset of $\overline{\mathcal{B}}$, it defines the value $v|_u^*$ which is an upper bound on the value attainable from that node.

As shown in Algorithm 10, the value $v|_u^*$ can prove useful at two different moments. First, in the event where $v|_u^* \leq \underline{v}$, this value can serve as a justification to not enqueue the subproblem u (lines 18–20) since exhausting this subproblem will yield no better solution than $\underline{v}$.

Algorithm 10 Branch-And-Bound with MDD with Local Bounds pruning

1: **Input:** a DP-model $\mathcal{P} = \langle S, r, t, \perp, v_r, \tau, h \rangle$
2: **Input:** a node merging operator $\oplus$
3: **Input:** an arc relaxation operator Γ
4: Create node r and add it to *Fringe*
5: $\underline{x} \leftarrow \perp$
6: $\underline{v} \leftarrow -\infty$
7: **while** *Fringe* is not empty **do**
8: $u \leftarrow Fringe.pop()$
9: **if** $v|_u^* \leq \underline{v}$ **then**
10: **continue**
11: $\underline{\mathcal{B}} \leftarrow Restricted(u)$
12: **if** $v^*(\underline{\mathcal{B}}) > \underline{v}$ **then**
13: $\underline{v} \leftarrow v^*(\underline{\mathcal{B}})$
14: $\underline{x} \leftarrow x^*(\underline{\mathcal{B}})$
15: **if** $\underline{\mathcal{B}}$ is not exact **then**
16: $\overline{\mathcal{B}} \leftarrow Relaxed(u, \oplus, \Gamma)$
17: **if** $v^*(\overline{\mathcal{B}}) > \underline{v}$ **then**
18: **for all** $u' \in \overline{\mathcal{B}}.exact_cutset()$ **do**
19: **if** $v|_{u'}^* > \underline{v}$ **then**
20: $Fringe.add(u')$
21: **return** $(\underline{x}, \underline{v})$

More formally, by definition of a cutset and of LocB, it must be the case that the longest r-t path of $\overline{\mathcal{B}}$ traverses one of the cutset nodes u and thus that $v^*(\overline{\mathcal{B}}) = v|_u^*$ (where $v|_u^*$ is the local bound of u). Hence we have: $\exists u \in$ cutset of $\overline{\mathcal{B}} : v^*(\overline{\mathcal{B}}) = v|_u^*$. However, because $v^*(\overline{\mathcal{B}})$ is the length of the *longest* r-t path of $\overline{\mathcal{B}}$, there may exist cutset nodes that only belong to r-t paths shorter than $v^*(\overline{\mathcal{B}})$. That is: $\forall u' \in$ cutset of $\overline{\mathcal{B}} : v^*(\overline{\mathcal{B}}) \geq v|_{u'}^*$. Which is why $v|_{u'}^*$ can be stricter than $v^*(\overline{\mathcal{B}})$ and hence let LocB be stronger at pruning nodes from the frontier.

The second time when $v|_u^*$ might come in handy occurs when the node u

is popped out of the fringe (lines 9–10). Indeed, because the fringe is a global priority queue, any node that has been pushed on the fringe can remain there for a long period of time. Thus, chances are that the value $\underline{v}$ has increased between the moment when the node was pushed onto the fringe (line 20) and the moment when it is popped out of it. Hence, this gives us an additional chance to completely skip the exploration of the sub-problem rooted in u.

Let us illustrate that with the relaxed MDD shown on Figure 5.1, for which the exact cutset comprises the highlighted nodes a and b. Please note that because this scenario may occur at any time during the problem resolution, we will assume that the fringe is not empty when it starts. Assuming that the current best solution $\underline{v}$ is 20 when one explores the pictured subproblem, we are certain that exploring the subproblem rooted in a is a waste of time, because the local bound $v|_a^*$ is only 16. Also, because the fringe was not empty, it might be the case that b was left on the fringe for a long period of time. And because of this, it might be the case that the best known value $\underline{v}$ was improved between the moment when b was pushed on the fringe and the moment when it was popped out of it. Assuming that $\underline{v}$ has improved to 110 when b is popped out of the fringe, it may safely be skipped because $v|_b^*$ guarantees that an exploration of b will not yield a better solution than 102.

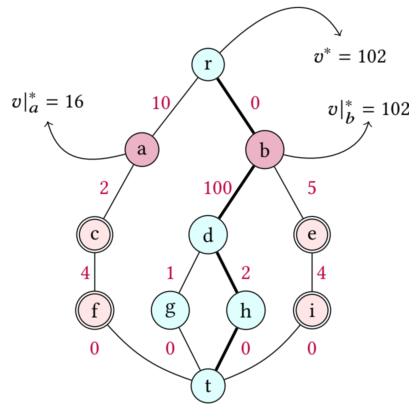


Figure 5.1: An example relaxed-MDD having an exact cutset $\{a, b\}$ with local bounds $v|_a^*$ and $v|_b^*$. The nodes with a simple border represent exact nodes and those with a double border represent "inexact" nodes. The edges along the longest path are displayed in bold.

Algorithm 11 describes the procedure to compute the local bound $v|_u^*$ of each node u belonging to the exact cutset of a relaxed MDD $\overline{\mathcal{B}}$. Intuitively, this is achieved by doing a bottom-up traversal of $\overline{\mathcal{B}}$, starting at t and stopping when the traversal crosses the last exact layer (line 10). During that bottom-up traversal, the algorithm marks the nodes that are reachable from

t. This way, it can avoid the traversal of dead-end nodes. Also, Algorithm 11 maintains a value $v^*_{\uparrow_t}(u)$ for each node u it encounters. This value represents the length of the longest u-t path. Afterwards (line 18), it is summed with the length of the longest r-u path v^*_{r-u} to derive the exact value of the local bound $v|^*_u$.

Algorithm 11 Computing the local bounds

```
1:  Input: B̄ a relaxed decision diagram
2:  lel ← Index of the last exact layer of B̄
3:  // initialize longest u-t path of all nodes
4:  for each node u ∈ B̄ do
5:      v*_{↑t}(u) ← −∞
6:  // initialize bottom-up traversal
7:  mark(t) ←true
8:  v*_{↑t}(t) ← 0
9:  // Actually traverse B̄ bottom-up
10: for all i = n to lel do
11:     for all node u ∈ L_i do
12:         if mark(u) then
13:             for all arc a = (u′, u, d) incident to u do
14:                 mark(u′) ← true
15:                 v*_{↑t}(u′) ← max(v*_{↑t}(u′), v*_{↑t}(u) + v(a))
16: for all node u ∈ B̄.exact_cutset() do
17:     if mark(u) then
18:         v|*_u ← v*_{r−u} + v*_{↑t}(u)
19:     else
20:         v|*_u ← −∞
```

5.1.1 Local Bound will not work with a shallow representation

As explained in Chapter 4, a shallow MDD representation is one that only maintains a "slice" of the actual DD it stands for[1]. All the previous layers are forgotten and the nodes belonging to the "slice" only need to keep track of their single best parent in order to be able to recover the solution maximizing the objective function. This is a perfectly sound and memory efficient way of representing MDDs when implementing a branch-and-bound DDO solver. However, this memory efficiency comes at a price: precious information is lost when the older layers are forgotten. And this information loss is precisely what prevents a shallow MDD representation to exploit local bounds.

[1]Flat MDD and Pooled MDD

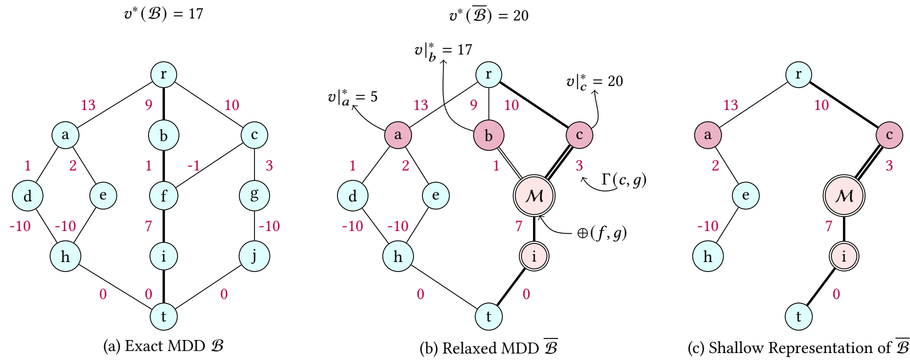


Figure 5.2: Example why local bounds cannot be implemented with a shallow MDD. (a) Displays the complete graph of the exact MDD $\mathcal{B}$. (b) Shows the $\overline{\mathcal{B}}$ which is a valid relaxation of $\mathcal{B}$. (c) Visually represents the information available after a shallow MDD representing $\overline{\mathcal{B}}$ finished unrolling. The typographic conventions are the same as for the previous figures: the longest path is boldfaced. Inexact nodes and relaxed arcs are double-stroked. The background of the nodes from the exact cutset is red.

The example from Figure 5.2 illustrates why such an implementation would not be correct. On the left (a), Figure 5.2 pictures the exact MDD $\mathcal{B}$ which comprises five layers. The maximum width of $\mathcal{B}$ is 4 because its third layer comprises the nodes d, e, f and g. The optimum value $v^*(\mathcal{B})$ of this DD is 17, and it is reached by following the path (r, b, f, i, t). The center of Figure 5.2 (b) depicts the relaxed MMD $\overline{\mathcal{B}}$ corresponding to a relaxation of $\mathcal{B}$ with a maximum width of 3. Because of the maximum width constraint enforced on $\overline{\mathcal{B}}$, the nodes f and g have been merged into $\mathcal{M}$ and their inbound arcs have been relaxed with the Γ operator. The best value $v^*(\overline{\mathcal{B}})$ of this DD amounts to 20 because it is the length of its longest path $(r, c, \mathcal{M}, i, t)$. Its LEL cutset is made of the nodes a, b and c. Finally, the right (c) of the Figure 5.2 shows the information available after a shallow MDD finished unrolling $\overline{\mathcal{B}}$. Because it is a shallow representation, the nodes only know of their "best" parent. That is, their direct ancestor on to the longest path between them and t. In that configuration, it is impossible for the node b to ever be marked or acquire a local bound $v|_b^*$ other than $-\infty$ after an execution of Algorithm 11. This occurs because all b-t paths have been forgotten by the shallow representation. Hence, if one were to use local bounds computed as stated above, the node b would be pruned off the exact cutset and never make its way to the solver fringe. Because of that, the solver would fail to identify the optimal solution of $\mathcal{B}$ because its single longest path necessarily goes through b.

5.1.2 Complexity matters...

As shown in Algorithm 11, the computation of the local bound $v|_u^*$ of some node u from the exact cutset of a relaxed MDD $\overline{\mathcal{B}}$ only requires the ability to remember if u is reachable from t ($mark(u)$) and to sum the lengths of the longest r-u path v_{r-u}^* and the longest u-t path $v_{\uparrow_t}^*(u)$. To that end, it is sufficient to only store one flag and two integers in each node. Thus, the spatial complexity of implementing local bounds is $\Theta(1)$ per node.

Similarly, the backward traversal of $\overline{\mathcal{B}}$ as performed in Algorithm 11 only visits a subset of the nodes and arcs that have been created during the compilation of $\overline{\mathcal{B}}$. The time complexity of Algorithm 11 is thus bounded by $O(|U| \times |A|)$. Actually, we even know that this bound is lesser or equal to that of the top-down compilation because the width-bounding procedure reduces the number of nodes in $\overline{\mathcal{B}}$. Hence it reduces the number of nodes potentially visited by Algorithm 11. It follows that the use of local bounds has no impact on the time complexity of the derivation of a relaxed MDD.

5.2 Rough upper bound (RUB)

Rough Upper Bound pruning is a new rule to reduce the search space that needs to be developed during the compilation of a bounded width MDD. It departs from the following observation: assuming the knowledge of a lower bound $\underline{v}$ on the global optimum v^*, and assuming that one is able to swiftly compute a rough upper bound $\overline{v_s}$ on the optimal value v_s^* of the subproblem rooted in state s; any node u of a MDD having a rough upper bound $\overline{v_{\sigma(u)}} \leq \underline{v}$ may be discarded as it is guaranteed not to improve the best known solution. Fundamentally, this is the exact same reasoning that underlies the whole branch-and-bound idea. But here, it is used to prune portions of the search space explored *while compiling* approximate MDDs.

As shown per Algorithm 12, in order to implement RUB, it suffices to adapt the MDD compilation procedure and introduce a check that avoids creating a node u' with state $next$ when $\overline{v_{next}} \leq \underline{v}$ (Algorithm 12 line 10).

The key to RUB effectiveness is that RUB is used while compiling the restricted and relaxed DDs. As such, its computation does not directly appear in the branch-and-bound with MDD algorithm (Algorithm 5, Chapter 2 p. 18). Instead, it is accounted within the compilations of $Restricted(u)$ and $Relaxed(u)$ from that same algorithm. Thus, it really is not used as yet another bound competing with those of the bounded width DDs, but instead as a means to speed up the compilation of these very DDs. More precisely, this speedup occurs because the compilation of the DDs discards some nodes that would otherwise be added to the next layer of the DD and then further expanded, which are ruled out by RUB. A second benefit of using RUBs is that

Algorithm 12 Top Down Compilation of a Bounded-Width DD with RUB

```
1:  Input: a DP-model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  Input: a maximum layer width W
3:  Input: a node merging operator ⊕
4:  Input: an arc relaxation operator Γ
5:  Input: a lower bound v̲ on the global optimum v*
6:  L_0 ← {r}
7:  for i ∈ {0 . . . n − 1} do
8:     for u ∈ L_i, d ∈ D_i do
9:        u′ ← a node associated with state τ_i(σ(u), d)
10:       if σ(u′) ≠ ⊥ ∧ rub(u′) > v̲ then
11:          U ← U ∪ {u′}
12:          L_{i+1} ← L_{i+1} ∪ {u′}
13:          a ← (u, u′, d)
14:          v(a) ← h_i(σ(u), d)
15:          A ← A ∪ {a}
16:    if |L_{i+1}| > W then
17:       L_{i+1} ← squash_layer(L_{i+1}, U, A, ⊕, Γ)
```

it helps tightening the bound derived from a relaxed DD. Because the layers that are generated in a relaxed DD are narrower when applying RUB, there are fewer nodes exceeding the maximum layer width. The operator $\oplus$ hence needs to merge a smaller set of nodes in order to produce the relaxation. It is therefore less likely to include a node belonging to the longest r-t path in the relaxation – which could have a negative effect on the tightness of the bound. Doing so, it maximizes the chances of deriving as tight a bound when compiling a relaxed DD.

The dynamics of RUB is graphically illustrated by Figure 5.3 where the set of highlighted nodes can be safely elided since the (rough) upper bound computed in node s is lesser than the best lower bound.

Important Note It is important to understand that because the RUB is computed at each node of each restricted and relaxed MDD compiled during the instance resolution, it must be extremely inexpensive to compute. This is why RUB is best obtained from a fast and simple problem specific procedure.

5.2.1 Rough Upper Bounds for the Applications problems

In Chapter 3 of the present thesis, detailed DP models have been presented for the Maximum Independent Set Problem (MISP), the Maximum Cut Problem (MCP), the Maximum 2-Satisfiability Problem (MAX2SAT), the Travel-

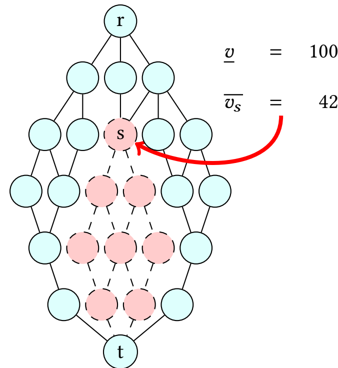


Figure 5.3: Assuming a lower bound $\underline{v}$ of 100 and a rough upper bound $\overline{v_s}$ of 42 for the node s, all the highlighted nodes (in red, with a dashed border) may be pruned from the MDD.

ing Salesman Problem with Time Windows (TSPTW) and the Pigment Sequencing Problem (PSP). The following sections establish a bridge between Chapters 3 and 5. It introduces a rough upper bound (rough lower bound for minimization problems) for each of the 5 considered problems.

5.2.1.1 RUB for the MISP

Assuming the MISP DP formulation presented in Chapter 3, where each state s from the DP model is a set of vertices that might possibly take part in a maximum independent set *given the decisions labeling the path between the root state r and s.* A simple over approximation of the objective value for the subproblem rooted in s considers that all vertices $v \in s$ having a positive weight will eventually take part in the maximum independent set.

Obviously, this is a very optimistic approach and it is highly unlikely to yield the actual optimum. Still, this approach produces a correct over approximation and a fast procedure is easily designed to compute that value.

Formally, the RUB $\overline{v^s}$ of the subproblem rooted in s is expressed by:

$$\overline{v^s} = v^*(s) + \sum_{i \in s} (w_i)^+ \tag{5.1}$$

where $v^*(s)$ is the length of the longest $r - s$ path, and $\sum_{i \in s} (w_i)^+$ is the sum of the positive weights[2] of vertices in state s.

Theorem 5.2.1 *Equation (5.1) is an upper bound (admissible heuristic) for the MISP. In other words, the length $v|_s^*$ of the longest r-t path passing through s is shorter or equal to the proposed RUB $\overline{v^s}$.*

[2] In line with in Chapter 3, the notation $(\alpha)^+$ is used as a shorthand form for $\max\{\alpha, 0\}$.

$$v|_s^* \leq \overline{v^s}$$

Proof 5.2.1 *The correctness of this RUB is trivially proved by contradiction. To that end, let us call α the optimal solution for the MISP subproblem rooted in s. On that basis, the length $v|_s^*$ of the longest r-t path passing through s can be rewritten as:*

$$v|_s^* = v^*(s) + f(\alpha) \tag{5.2}$$

$$= v^*(s) + \sum_{i \in \alpha} w_i \tag{5.3}$$

$$= v^*(s) + \sum_{i \in \alpha} (w_i)^+ \tag{5.4}$$

where 5.3 follows from the transition function and 5.4 is guaranteed because MISP only consider positive weights (failing to do so would decrease the objective value). In order to establish Theorem 5.2.1, it suffice to prove the impossibility of:

$$v^*(s) + \sum_{i \in \alpha} (w_i)^+ > v^*(s) + \sum_{i \in s} (w_i)^+$$

That is, it suffices to prove the impossibility of

$$\sum_{i \in \alpha} (w_i)^+ > \sum_{i \in s} (w_i)^+$$

By definition of the MISP, we have $\alpha \subseteq s$ which makes the above impossible to satisfy since $\sum_{i \in \alpha} (w_i)^+$ and $\sum_{i \in s} (w_i)^+$ add positive terms only. Satisfying $\sum_{i \in \alpha} (w_i)^+ > \sum_{i \in s} (w_i)^+$ would thus require that α contain at least one vertex with a positive weight more than s. This requirement contradicts the definition of α as the optimal solution to the subproblem rooted in s.

∎

5.2.1.2 RUB for the MCP

The MCP model detailed in Chapter 3 is not trivial and defines its states as n-tuples where each component of s_i^k of a given state s^k represents the marginal benefit of assigning vertex i to partition T based on the first k decisions. Still, a rough upper bound for that model can be formulated as follows. Formally, the RUB $\overline{v^{s^k}}$ of the subproblem rooted in state s^k is expressed by:

$$\overline{v^{s^k}} = v^*(s^k) + \sum_{k \leq i < n} |s_i^k| + \mathcal{V}_k + \mathcal{N}_k - v_r \tag{5.5}$$

where s^k is a state from the kth layer, $v^*(s^k)$ is the length of the longest $r - s^k$ path and v_r is the root value of the DP model. $\mathcal{V}_k = \sum_{k \le i < j < n} (w_{ij})^+$ is an over estimation of the maximum cut value on the remaining induced graph (sum of the weight of all the positive arcs in the induced graph). $\mathcal{N}_k = \sum_{0 \le i < j < k} (w_{ij})^-$ is the partial sum of the weights of all negative arcs interconnecting vertices which have already been assigned to a partition.

This upper bound is very fast to compute – amortized $O(n)$ – since $v^*(s^k)$ is known when node s^k must be expanded and the quantities v_r, $\mathcal{V}_k$ and $\mathcal{N}_k$ can be pre-computed once for all (assuming a fixed variable ordering).

The correctness of this RUB is proved by showing that, for any given state s^k, the bound which is computed is either larger or equal to the longest $r - t$ path passing through state s^k.

Theorem 5.2.2 *The length $v|^*_{s^k}$ of the longest r-t path passing through s^k is shorter or equal to the RUB presented above ($\overline{v^{s^k}}$)*

$$v|^*_{s^k} \le \overline{v^{s^k}}$$

Proof 5.2.2 *In order to compute the length $v|^*_{s^k}$ of the longest $r - t$ path passing through s^k, one essentially needs to compute the sum of:*

1. *the longest $r - s^k$ path in the subproblem $\mathcal{P}_{0,k}$ bearing only on variables for which a decision has been made $[0 \dots k - 1]$. The length of this path is denoted $v^*_{\mathcal{P}_{0,k}}(s^k)$. Put another way, this quantity represents the sum of weights of edges which are fully determined to cross the cut. (Both ends have been assigned to a partition).*

2. *the value of the best solution to the subproblem $\mathcal{P}_{k,n}$ bearing only on variables for which no decision has been made $[k \dots n - 1]$. We use $v^*(\mathcal{P}_{k,n})$ to denote that value. In other words, $v^*(\mathcal{P}_{k,n})$ is the weight of the maximum cut in the residual graph comprising only nodes that have not been assigned to either partition S or T.*

3. *the value C_k of the maximum cut considering only edges for which exactly one end is a vertex that has been assigned to a partition during the first k decisions.*

This lets us rewrite $v|^*_{s^k}$ as shown below:

$$v|^*_{s^k} = v^*_{\mathcal{P}_{0,k}}(s^k) + v^*(\mathcal{P}_{k,n}) + C_k, \forall 0 \le k < n \tag{5.6}$$

According to our decomposition methodology, we would like to start by showing that

$$\begin{aligned} v^*_{\mathcal{P}_{0,k}}(s^k) &= v^*(s^k) - v_r + \mathcal{N}_k \\ v_r(\mathcal{P}_{0,k}) + \sum_{0\le i<k} h(s^i, x_i) &= v^*(s^k) - v_r + \mathcal{N}_k \\ \sum_{0\le i<j<k} (w_{ij})^- + \sum_{0\le i<k} h(s^i, x_i) &= v^*(s^k) - v_r + \mathcal{N}_k \\ \mathcal{N}_k + \sum_{0\le i<k} h(s^i, x_i) &= v^*(s^k) - v_r + \mathcal{N}_k \\ \mathcal{N}_k + \sum_{0\le i<k} h(s^i, x_i) &= v_r + \sum_{0\le i<k} h(s^i, x_i) - v_r + \mathcal{N}_k \\ \mathcal{N}_k + \sum_{0\le i<k} h(s^i, x_i) &= \sum_{0\le i<k} h(s^i, x_i) + \mathcal{N}_k \end{aligned} \tag{5.7}$$

Then, we show that

$$v^*(\mathcal{P}_{k,n}) \le \mathcal{V}_k \tag{5.8}$$

By definition of $v^*(\mathcal{P}_{k,n})$ *as the value of the maximum cut in the residual graph comprising only vertices not assigned to any partition, we have:* $v^*(\mathcal{P}_{k,n}) = \max \sum_{k\le i<j<n} w_{ij}(x_i \neq x_j)$. *From there, it follows that:*

$$\begin{aligned} v^*(\mathcal{P}_{k,n}) &\le \mathcal{V}_k \\ \max \sum_{k\le i<j<n} w_{ij}(x_i \neq x_j) &\le \mathcal{V}_k \\ \max \sum_{k\le i<j<n} w_{ij}(x_i \neq x_j) &\le \sum_{k\le i<j<n} (w_{ij})^+ \end{aligned}$$

Finally, we show that

$$C_k \le \sum_{k\le i<n} |s^k_i| \tag{5.9}$$

which is proved by contradiction. Indeed, falsifying that property would require that $C_k > \sum_{k\le i<n} |s^k_i|$. *That is, it would require that the value of the maximum cut comprising only edges having exactly one end assigned to some partition (either* S *or* T*) be strictly greater than the sum of the marginal benefit for the "perfect" assignment for these edges. This either contradicts the definition of a maximum cut (hence the definition of* C_k*) or that of the state transition function.*

Combining (5.7), (5.8) and (5.9); we have:

$$\left(v^*_{\mathcal{P}_{0,k}}(s^k) = v^*(s^k) - v_r + \mathcal{N}_k\right) \wedge \left(v^*(\mathcal{P}_{k,n}) \le \mathcal{V}_k\right) \wedge \left(C_k \le \sum_{k\le i<n} |s^k_i|\right)$$
$$\Longrightarrow$$
$$v^*_{\mathcal{P}_{0,k}}(s^k) + v^*(\mathcal{P}_{k,n}) + C_k \le v^*(s^k) + \sum_{k\le i<n} |s^k_i| + \mathcal{V}_k + \mathcal{N}_k - v_r$$

Which can be reformulated as follows using (5.6) and (5.5):

$$\left(v^*_{\mathcal{P}_{0,k}}(s^k) = v^*(s^k) - v_r + \mathcal{N}_k\right) \wedge \left(v^*(\mathcal{P}_{k,n}) \leq \mathcal{V}_k\right) \wedge \left(C_k \leq \sum_{k \leq i < n} |s^k_i|\right)$$
$$\Longrightarrow$$
$$v|^*_{s^k} \leq \overline{v^{s^k}}$$

Also, because all terms on the left hand side of the implication have been proved true in equation (5.7), (5.8) and (5.9); we have:

$$v|^*_{s^k} \leq \overline{v^{s^k}}$$

■

5.2.1.3 RUB for MAX2SAT

The DP model and relaxation that have been presented in Chapter 3 for MAX2-SAT resemble those of MCP. In both cases, the states are tuples of marginal costs associated with one of two opposite decisions. Moreover, the definition and interpretation of the relaxation operators are identical in both cases. The difference between these models essentially stems from the peculiarities of each model, and more specifically what it means to "acquire" a cost (edge crossing a cut for MCP versus satisfying at least one literal in a clause for MAX2SAT). Given these similarities, it is only natural that the RUBs that have been designed for both problems exhibits some similarities as well since the general approach is kept across both problem. Still, the details of both RUBs vary. Given a MAX2SAT state s^k, a RUB can be computed as shown below:

$$\overline{v^{s^k}} = v^*(s^k) + \sum_{k \leq i < n} |s^k_i| + \mathcal{V}_k \tag{5.10}$$

where $v^*(s^k)$ is the length of a longest $r - s^k$ path, $\sum_{k \leq i < n} |s^k_i|$ is the total marginal gain that can possibly be earned based on previous decisions; and $\mathcal{V}_k$ is an over approximation on the value of the residual subproblem[3] bearing

[3] As it is expressed here, this over approximation does not account for the tautologies in that residual subproblem. These could have been included at the expense of making equation (5.10) slightly more complex. In which case, the formulation of the RUB for MAX2SAT would have had the exact same form as the MCP RUB (albeit with different definitions of sub terms). However, including the cost of tautologies is not required in when defining this RUB because their cost is already accounted for by the initial value v_r of the DP model. Which means, the cost of tautologies is readily covered in $v^*(s^k)$ and does not need to be explicitly mentioned in (5.10).

on undecided variables only. Specifically, we have

$$\mathcal{V}_k = \sum_{k \le i < n} \max\left\{w_{i,i}^{TT}, w_{i,i}^{FF}\right\} \tag{5.11}$$

$$+ \sum_{k \le i < j < n} \max\left\{g_{i,j}^{TT}, g_{i,j}^{TF}, g_{i,j}^{FT}, g_{i,j}^{FF}\right\} \tag{5.12}$$

where (5.11) is the sum of the weight of all unit clauses that might possibly be satisfied by the residual sub problem, and (5.12) over approximates the possible total gain earned of deciding all remaining pairs of undecided variables. In this definition, the expressions $g_{i,j}^{TT}, g_{i,j}^{TF}, g_{i,j}^{FT}$ and $g_{i,j}^{FF}$ are defined as:

$$g_{i,j}^{TT} = w_{i,j}^{TT} + w_{i,j}^{TF} + w_{i,j}^{FT} \tag{5.13}$$

$$g_{i,j}^{TF} = w_{i,j}^{TT} + w_{i,j}^{TF} + w_{i,j}^{FF} \tag{5.14}$$

$$g_{i,j}^{FT} = w_{i,j}^{TT} + w_{i,j}^{FT} + w_{i,j}^{FF} \tag{5.15}$$

$$g_{i,j}^{FF} = w_{i,j}^{TF} + w_{i,j}^{FT} + w_{i,j}^{FF} \tag{5.16}$$

Intuitively, $g_{i,j}^{TT}$ is the gain accrued by a joint assignment of variables x_i and x_j where both variables are set to true $[\![x_i = T, x_j = T]\!]$. It sums the weight of clauses $(x_i \vee x_j)$, $(x_i \vee \neg x_j)$, and $(\neg x_i \vee x_j)$ but it purposely omits the weight of clause $(\neg x_i \vee \neg x_j)$ as that one clause is known to be falsified by the joint assignment (5.13). Similarly, (5.14) defines $g_{i,j}^{TF}$ as the accrued gain of the joint assignment $[\![x_i = T, x_j = F]\!]$, (5.15) defines $g_{i,j}^{FT}$ as the gain of the joint assignment $[\![x_i = F, x_j = T]\!]$, and (5.16) defines $g_{i,j}^{FF}$ as the gain of the joint assignment $[\![x_i = F, x_j = F]\!]$.

The proof of the correctness of this RUB is analogous to the one detailed for MCP. In order to effectively prove the correctness of this bound, it suffices to prove that for any state s^k, the RUB $\overline{v^{s^k}}$ is larger or equal to the longest $r - t$ path passing through state s^k.

Theorem 5.2.3 *The length $v|_{s^k}^*$ of the longest r-t path passing through s^k is shorter or equal to the RUB presented above ($\overline{v^{s^k}}$)*

$$v|_{s^k}^* \le \overline{v^{s^k}}$$

Proof 5.2.3 *Computing the length of the longest r-t path passing through s^k amounts to computing the sum of:*

- *The cost the longest path in the subproblem $\mathcal{P}_{0,k}$ bearing only on variables for which a decision has already been made. The length of this path is denoted $v_{\mathcal{P}_{0,k}}^*(s^k)$. Put another way, this quantity represents the sum of weights of all clauses that are irremediably satisfied based on the current partial assignment.*

- *The value of the best solution to the residual subproblem $\mathcal{P}_{k,n}$ bearing on unassigned variables only. This value will be referred to as $v^*(\mathcal{P}_{k,n})$ and*

- *The weight C_k of all the clauses that have not been satisfied by the first k assignments but are satisfied thanks to a decision made along the longest path.*

This decomposition lets us rewrite $v|^*_{s^k}$ as shown below:

$$v|^*_{s^k} = v^*_{\mathcal{P}_{k,n}}(s^k) + v^*(\mathcal{P}_{k,n}) + C_k, \forall 0 \leq k < n \tag{5.17}$$

In addition to the above, let us define the term $\mathcal{N}_k$ as the sum of the weights of tautologies bearing on assigned variables only:

$$\mathcal{N}_k = \sum_{0 \leq i < k} w^{TF}_{i,i} \tag{5.18}$$

From that definition, it follows that:

$$v^*_{\mathcal{P}_{0,k}}(s^k) = v^*(s^k) - v_r + \mathcal{N}_k \tag{5.19}$$

We then establish that

$$v^*(\mathcal{P}_{k,n}) - \sum_{k \leq i < n} w^{TF}_{i,i} \leq \mathcal{V}_k \tag{5.20}$$

which is obvious since $\mathcal{V}_k$ is the maximum weight of the clauses of $\mathcal{P}_{k,n}$ that can be satisfied by a (possibly inconsistent) set of pairwise assignment. And because the subtracted term $\sum_{k \leq i < n} w^{TF}_{i,i}$ removes the cost of tautologies which are not covered by $\mathcal{V}_k$.

Finally, we show that

$$C_k \leq \sum_{k \leq i < n} |s^k_i| \tag{5.21}$$

which is proved by contradiction. Falsifying that condition would required that $C_k > \sum_{k \leq i < n} |s^k_i|$. In other words, falsifying this condition would require that the total weight of the clauses whose truth value depend on one single unassigned literal be greater than the total marginal benefit for a "perfect" assignment of these literals. This requirement contradicts the definition of C_k and hence establishes the truth of (5.21).

Combining (5.17) with (5.18), (5.19), (5.20) and (5.21); we have:

$$\begin{aligned}
v|^*_{s^k} &= s^*_{\mathcal{P}_{0,k}}(s^k) + v^*(\mathcal{P}_{k,n}) + \mathcal{C}_k \\
&= v^*(s^k) - v_r + \mathcal{N}_k + v^*(\mathcal{P}_{k,n}) + \mathcal{C}_k \\
&= v^*(s^k) - \sum_{0 \le i < n} w^{TF}_{i,i} + \sum_{0 \le i < k} w^{TF}_{i,i} + v^*(\mathcal{P}_{k,n}) + \mathcal{C}_k \\
&\le v^*(s^k) - \sum_{0 \le i < n} w^{TF}_{i,i} + \sum_{0 \le i < k} w^{TF}_{i,i} + \mathcal{V}_k + \sum_{k \le i < n} w^{TF}_{i,i} + \mathcal{C}_k \\
&\le v^*(s^k) + \mathcal{V}_k + \mathcal{C}_k \\
&\le v^*(s^k) + \mathcal{V}_k + \sum_{k \le i < n} |s^k_i| \\
&\le \overline{v^{s^k}}
\end{aligned}$$

■

5.2.1.4 RLB for the TSPTW

> **Note**
>
> **Reminder** TSPTW is a minimization problem. It must thus define a rough *lower* bound rather than a rough *upper* bound.

In Chapter 3, a DP model for TSPTW was presented where a state s^k from the kth layer it a tuple $\langle time^k, position^k, must_visit^k, might_visit^k \rangle$. In that context, a simple lower bound would consist of the total weight of a minimum spanning tree covering all the cities that *must* still be visited. While this bound is correct – it is the usual lower bound for TSP-style problems, we observed experimentally that the complexity $O(n^2 \; log \; n)$ with n being the number of cities, was too expensive in practice for it to be an effective RLB.

We therefore devised a new bound which is slightly more involved but turns out to be a more effective RLB. This new RLB is easily computed in $O(n \; log \; n)$. Essentially, our bounding accounts for five different costs that are required to complete a valid TSPTW tour. As illustrated per Figure 5.4, these five costs can be summed together and respectively correspond to:

1. The total distance that has already been traveled ($v^*(s^k)$).
2. The distance between $position^k$ and any remaining node.
3. An estimate of the distance between the nodes that must be visited.

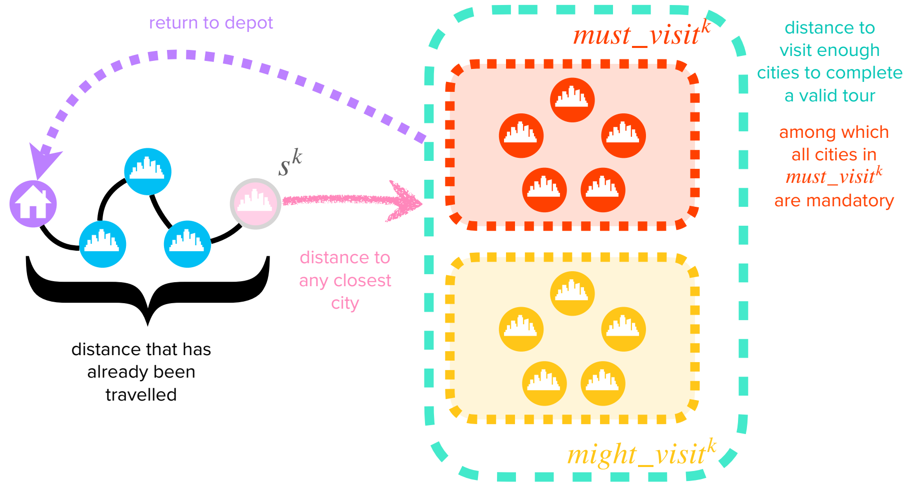


Figure 5.4: Illustration of the 5 costs required to complete a valid TSPTW tour from a given state s^k.

4. An estimate of the distance required to visit $n - k - |must_visit^k|$ cities among those which might be visited (without violating time windows).

5. An estimate of the distance required to head back to depot.

As it has been presented until now, it appears as though (2) and possibly (3) might involve computationally expensive operations similar to computing a minimum spanning tree (MST). However, this turns out to not be the case. Indeed, a important ingredient of our RLB is the use of an additional vector *cheapest* which stores for each city i the cost of its cheapest adjacent edge. This vector is precomputed before to starting actually solving an instance.

In practice, $\sum_{0 \le i < n} cheapest_i$ is the cost of an MST at root state. However, the bound it provides at later levels is much less tight since the cheapest edge adjacent to some city i might not be a considerable option.

Distance from current position Nevertheless, using the *cheapest* vector allows the computation of an inexpensive estimate of the distance between the current position and any other city.

$$est_1 = \min \left\{ cheapest_i \mid i \in position^k \right\}$$

Distance between the cities that must be visited Similarly, an estimate of the distance between all the cities that must be visited consists of the sum of their cheapest edges $\sum i \in must_visit^k cheapest_i$.

However, this bound can easily be strengthened by including a feasibility check. In case an infeasibility is detected, then our RLB returns $+\infty$. The cost associated to each city when computing this estimate is thus given by:

$$feasible_distance_i^k = \begin{cases} cheapest_i & \text{if } time^k + est_1 \leq l_i \\ +\infty & \text{otherwise} \end{cases}$$

On that basis, the second term intervening in our RLB is computed as:

$$est_2 = \sum_{i \in must_visit^k} feasible_distance_i^k$$

Estimate of the time it takes to visit n cities in the tour A very crude estimate of the time it takes to visit the required number of cities in a tour consists of taking the cost of the $n - k - |must_visit^k|$ cheapest edges adjacent to nodes that *might* be visited. In other words, it completes est_2 to form an under approximation of an MST covering the required number of cities among which the cities in the set $must_visit^k$ are imposed.

In this case again, a feasibility check can be applied. Indeed, if the size of the set $\left\{i \in might_visit^k \mid time^k + est_1 \leq l_1\right\}$ is less than $n - k - |must_visit^k|$, then it is impossible to complete a valid tour. Therefore, our RLB returns $+\infty$ in this case as well.

Returning to depot The estimate of the cost it takes to return to the depot after all due cities have been visited is quite straightforward. It simply consists of the smallest distance between any of the visitable cities and the depot. Formally, it is given by

$$\min \left\{\mathcal{D}_{i,0} \mid i \in \left(must_visit^k \cup might_visit^k\right)\right\}$$

5.2.1.5 RLB for the PSP

Because PSP is a simple case of Wagner-Whitin (WW) [PW06] in the absence of changeover costs, the WW cost of some state $s^t = \langle k, u \rangle \in \mathcal{S}_{\mathcal{H}-t}$ is an admissible lower bound on the sub-problem rooted in s^t. Also, the PSP is essentially a TSP when the stocking cost and delivery constraints are lifted. Hence, the cost of an MST covering the changeover cost of all the item types that are still to be produced is also an admissible lower bound to this problem. Moreover, because the number of items $|\mathcal{I}|$ of PSP instances is usually small (less than 10), the cost of all these MST can be easily precomputed before to actually start solving the problem. The repeated use of these precomputed costs allows the amortization of these computations.

Finally, because there is no overlap in the costs that are covered by the WW estimate and the MST estimate, we know that the sum of these estimates yields an admissible lower bound. This sum is thus constitutive of a valid RLB for the PSP.

5.3 Experimental Study

In order to evaluate the impact of the pruning techniques proposed above, we conducted a series of experiments on four problems. In particular, we conducted experiments on the Maximum Independent Set Problem (MISP), the Maximum Cut Problem (MCP), the Maximum Weighted 2-Satisfiablility Problem (MAX2SAT), and the Traveling Salesman Problem with Time Windows (TSPTW). The DP models and benchmark instances we used for all problems are the ones that have been described in Chapter 3 of the present thesis. The *Rough Upper/Lower Bounds* that have been used are those presented in the previous section.

We attempted to solve each of problem instance with different configurations of *ddo*, our own open source solver written in Rust. Thanks to the generic nature of that framework, the model and all heuristics used to solve the instances were the same for all experiments. This allowed us to isolate the impact of LocB and RUB on the solving performance while neutralizing unrelated factors such as variable ordering. Indeed, the only variations between the different solver flavors relate to the presence (or absence) of the aforementioned pruning techniques. All experiments were run on the same physical machine equipped with an AMD6176 processor and 48GB of RAM. A maximum time limit of 30 minutes was allotted to each configuration to solve each instance.

Figures 5.5, 5.6, 5.7 and 5.8 give an overview of the results from our experimental study. They respectively depict the evolution over time of the number of instances solved by each technique for MISP (Figure 5.5), MCP (Figure 5.6), MAX2SAT (Figure 5.7) and TSPTW (Figure 5.8).

As a first step, our observation of the graphs will focus on the differences that arise between the single threaded configurations of our *ddo* solvers. Then, in a second phase, we will incorporate an existing state-of-the-art ILP solver (Gurobi 9.0.3) in the comparison. Also, because both Gurobi and our *ddo* library come with built-in parallel computation capabilities, we will consider both the single threaded and parallel (24 threads) cases. This second phase, however, only bears on MISP, MCP and MAX2SAT by lack of a Gurobi TSPTW model.

DDO configurations The first observation to be made about the four graphs is that for all considered problems, both RUB and LocB outperformed the 'do-

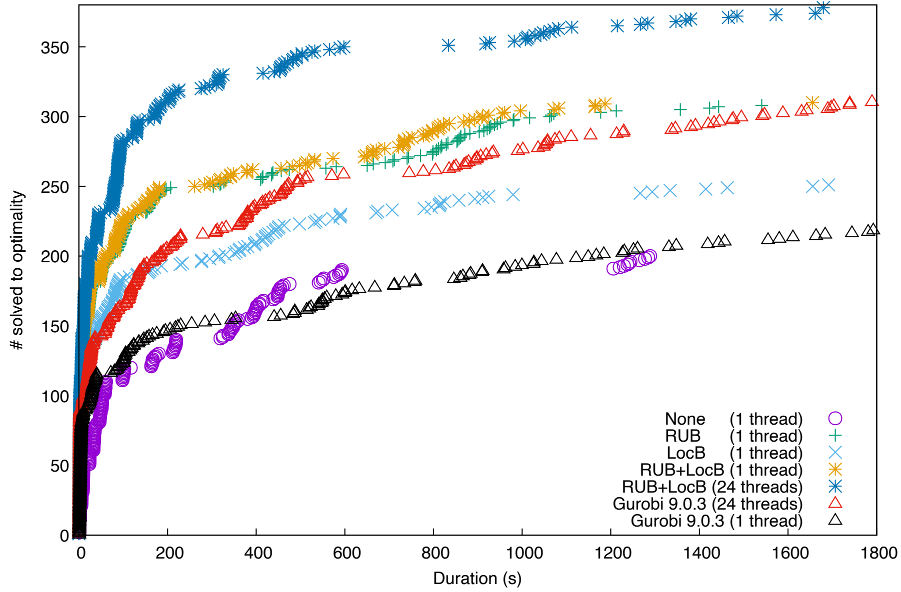


Figure 5.5: Number of solved instances over time for the Maximum Independent Set Problem (MISP)

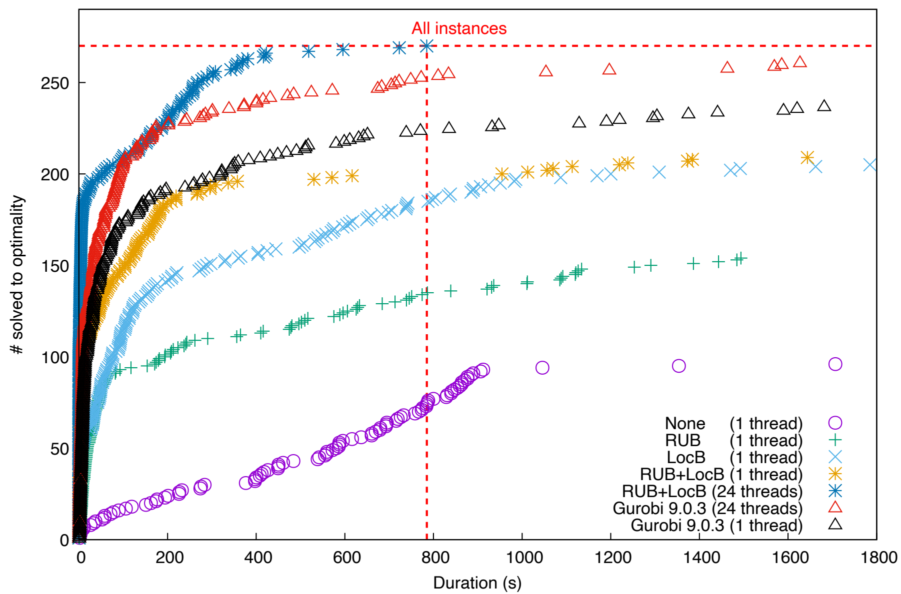


Figure 5.6: Number of solved instances over time for the Maximum Cutset Problem (MCP)

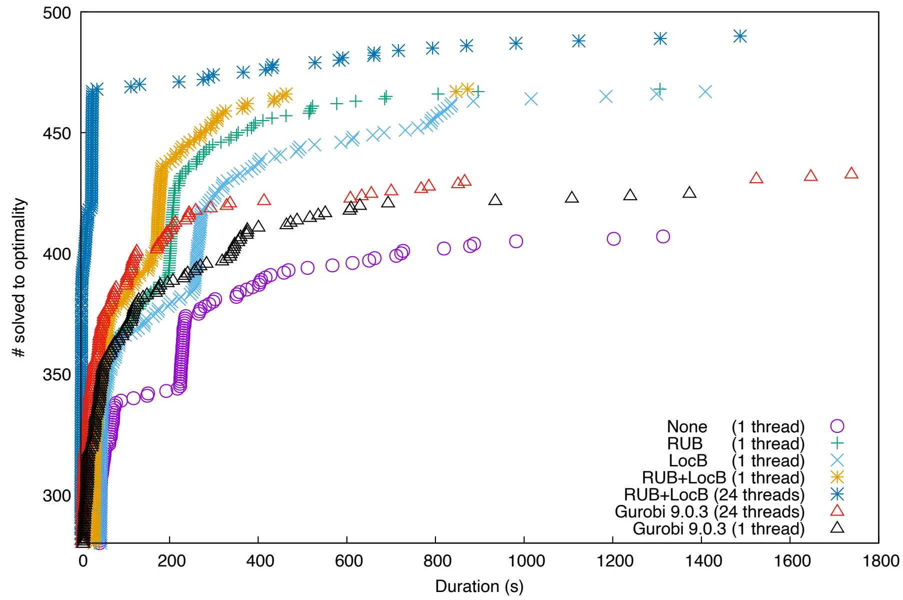


Figure 5.7: Number of solved instances over time for the Maximum 2 Satisfiablility Problem (MAX2SAT)

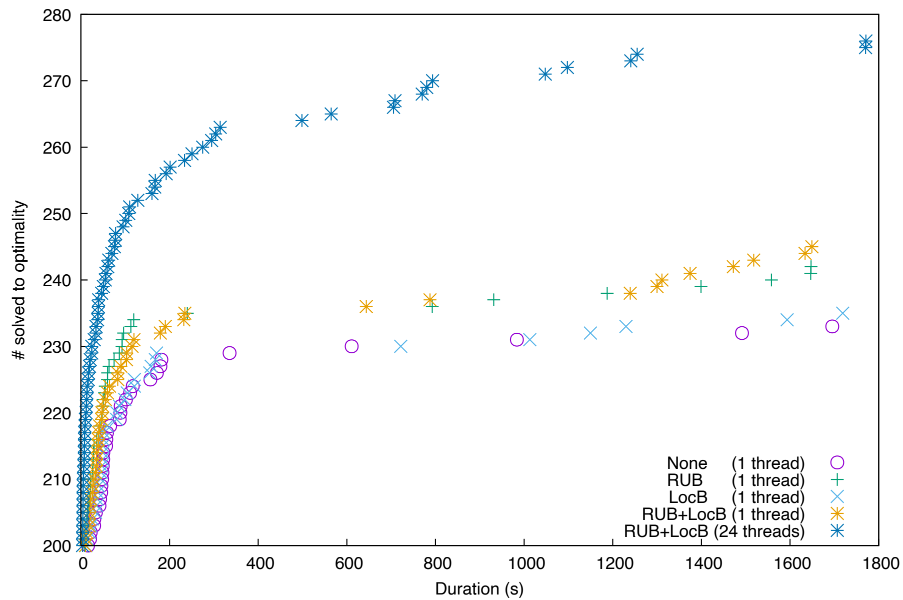


Figure 5.8: Number of solved instances over time for the Traveling Salesman with Time Windows Problem (TSPTW)

nothing' strategy; thereby showing the relevance of the rules we propose. It is not clear, however, which of the two rules brings the most improvement to the problem resolution. Indeed, RUB seems to be the driving improvement factor for MISP (a) and TSPTW (d) and the impact of LocB appears to be moderate or weak on these problems. However, it has a much higher impact for MCP (b) and MAX2SAT (c). In particular, LocB appears to be the driving improvement factor for MCP (b). This is quite remarkable given that LocB operates in a purely black box fashion, without any problem-specific knowledge. Finally, it should also be noted that the use of RUB and LocB are not mutually exclusive. Moreover, it turns out that for all considered problems, the combination RUB+LocB improved the situation over the use of any single rule.

Furthermore, Fig.5.9 confirms the benefit of using both RUB *and* LocB together rather than using any single technique. For each problem, it measures the "performance" of using RUB+LocB vs the best single technique through the end gap. The end gap is defined as $\left(100 * \frac{|UB|-|LB|}{|UB|}\right)$. This metric allows us to account for all instances, including the ones that could not be solved to optimality. Basically, a small end gap means that the solver was able to confirm a tight confidence interval of the optimum. Hence, a smaller gap is better. On each subgraphs of Fig.5.9, the distance along the x-axis represents the end gap for reach instance when using both RUB and LocB whereas the distance along y-axis represents the end gap when using the best single technique for the problem at hand. Any mark above the diagonal shows an instance for which, using both RUB and LocB helped reduce the end gap and any mark below that line indicates an instance where it was detrimental.

From graphs 5.9-a, 5.9-c and 5.9-d it appears that the combination RUB + LocB supersedes the use of RUB only. Indeed the vast majority of the marks sit above the diagonal and the rest on it. This indicates a beneficial impact of using both techniques even for the hardest (unsolved) instances. The case of MCP (graph 5.9-b) is less clear as most of the marks sit on the diagonal. Still, we can only observe three marks below the diagonal and a bit more above it. Which means that even though the use of RUB in addition to LocB is of little help in the case of MCP, its use does not degrade the performance for that considered problem.

Comparison with Gurobi 9.0.3 The first observation to be made when comparing the performance of Gurobi vs the DDO configurations, is that when running on a single thread, ILP outperforms the basic DDO approach (without RUB and LocB). Furthermore, Gurobi turns out to be the best single threaded solver for MCP by a fair margin. However, in the MISP and MAX2SAT cases, Figure 5.5 shows that the DDO solvers benefitting from

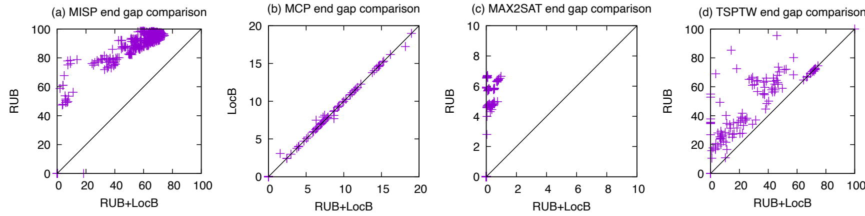


Figure 5.9: End gap: The benefit of using both techniques vs the best single one

RUB and LocB were able to solve more instances and to solve them faster than Gurobi. Which underlines the importance of RUB and LocB.

When lifting the one thread limit, one can see that the DD-based approach outperform ILP on each of the considered problems. In particular, in the case of MCP for which Gurobi is the best single threaded option; our DDO solver was able to find and prove the optimality of all tested instances in a little less than 800 seconds. The ILP solver, on the other end, was not able to prove the optimality of the 9 hardest instances within 30 minutes. Additionally, we also observe that in spite of the performance gains of MIP when running in parallel, Gurobi fails to solve as many MISP and MAX2SAT instances and to solve them as fast as the single threaded DDO solvers with RUB and LocB. This emphasizes once more the relevance of our techniques. It also shows that the observation from [Ber+14c] still hold today: despite the many advances of MIP the DDO approach still scales better than MIP on the considered problems when invoked in parallel.

5.4 Previous work

DDO emerged in the mid' 2000's when [Hoo06] proposed to use decision diagrams as a way to solve discrete optimization problems to optimality. More or less concomitantly, [And+07] devised relaxed-MDD even though the authors envisioned its use as a CP constraint store rather than a means to derive tight upper bounds for optimization problems. Then, the relationship between decision diagrams and dynamic programming was clarified by [Hoo13].

Recently, Bergman, Ciré and van Hoeve investigated the various ways to compile decision diagrams for optimization (top-down, construction by separation) [Cir14]. They also investigated the heuristics used to parameterize these DD compilations. In particular, they analyzed the impact of variable ordering in [Cir14; Ber+14b] and node selection heuristics (for merge and deletion) in [Ber+14b]. Doing so, they empirically demonstrated the crucial impact of variable ordering on the tightness of the derived bounds and highlighted the efficiency of minLP as a node selection heuristic. Later on, the

same authors proposed a complete branch-and-bound algorithm based on DDs [Ber+16b]. This is the algorithm which this chapter proposed to adapt with extra reasoning mechanisms and for which a generic open-source implementation in Rust [GSC20a]. The impressive performance of DDO triggered some theoretical research to analyze the quality of approximate MDDs [BC16b] and the correctness of the relaxation operators [Hoo17].

This gave rise to new lines of work. The first one focuses on the resolution of a larger class of optimization problems; chief of which multi-objective problems [BC16a] and problems with a non-linear objective function. These are either solved by decomposition [BC16a] or by using DDO to strengthen other IP techniques [DH18]. A second trend aims at hybridizing DDO with other IP techniques. For instance, by using Lagrangian relaxation [Hoo19] or by solving a MIP [BC17] to derive with very tight bounds. But the other direction is also under active investigation: for example, [Tja18; TH19] use DD to derive tight bounds which are used to replace LP relaxation in a cutting planes solver. Very recently, a third hybridization approach has been proposed by Gonzàlez et al.[Gon+20]. It adopts the branch-and-bound MDD perspective, but whenever an upper bound is to be derived, it uses a trained classifier to decide whether the upper bound is to be computed with ILP or by developing a fixed-width relaxed MDD.

The techniques (ILP-cutoff pruning and ILP-cutoff heuristic) proposed by Gonzalez et al.[Gon+20] are related to RUB and LocB in the sense that all techniques aim at reducing the search space of the problem. However, they fundamentally differ as ILP-cutoff pruning acts as a replacement for the compilation of a relaxed MDD whereas the goal of RUB is to speed up the development of that relaxed MDD by removing nodes *while* the MDD is being generated. The difference is even bigger in the case of ILP-cutoff heuristic vs LocB: the former is used as a primal heuristic while LocB is used to filter out sub-problems that can bear no better solution. In that sense, LocB belongs more to the line of work started by [And+07; HHT08; HVHH10]: it enforces the constraint $lb \leq f(x) \leq ub$ and therefore provokes the deletion of nodes and arcs that cannot lead to the optimal solution.

More recently, Horn et al. explored an idea in [Hor+21] which closely relates to RUB. They use “fast-to-compute dual bounds” as an admissible heuristic to guide the compilation of MDDs in an A* fashion for the prize-collecting TSP. It prunes portions of the state space during the MDD construction, similarly to when RUB is applied. Our approach differs from that of [Hor+21] in that we attempt to incorporate problem specific knowledge in a framework that is otherwise fully generic. More precisely, it is perceived here as a problem-specific pruning that exploits the combinatorial structure implied by the state variables. It is independent of other MDD compilation techniques, e.g. our techniques are compatible with node merge ($\oplus$) operators and other

methodologies defined in the DDO literature. We also emphasize that, as opposed to more complex LP-based heuristics that are now typical in A* search, we investigate quick methodologies that are also easy to incorporate in a MDD branch and bound.

5.5 Conclusion

This chapter presented and evaluated the impact of two reasoning rules (LocB and RUB) to strengthen the pruning of the branch-and-bound MDD algorithm. Our experimental study on MISP, MCP, MAX2SAT and TSPTW confirmed the relevance of these techniques. In particular, our experiments have shown that devising a fast and simple rough upper bound is worth the effort as it can significantly boost the efficiency of a solver. Similarly, our experiments showed that the use of local bound can significantly improve the efficiency of DDO solver despite its problem agnosticism. Furthermore, it revealed that a combination of both RUB and LocB supersedes the benefit of any single reasoning technique. These results are very promising and we believe that the public availability of an open source DDO framework implementing all these techniques might serve as a basis for novel DP formulation for classic problems.

Large Neighborhood Search with Decision Diagrams

6

Contributions and Publication Information

The content of this paper is largely based on Xavier Gillard and Pierre Schaus. *Large Neighborhood Search with Decision Diagrams.* IJCAI. 2022. The contribution of this chapter is a large neighborhood procedure which leverages restricted DD to swiftly explore good quality neighborhoods around a best known solution.

As opposed to previous chapters of this thesis, this chapter exits from the realm of exact solvers and proposes to blend DDO with heuristic methods and more specifically with a Large Neighborhood approach. This chapter shows how to design an efficient DD based neighborhood exploration reusing the idea of restricted DD introduced in [Ber+16b].

6.1 Motivation

Local search is a widely used approach to quickly obtain good solutions to combinatorial optimization problems [HM09; HS04]. Unfortunately, a simple gradient descent based on simple perturbations such as 2-exchange moves can quickly get trapped into a local minima. Metaheuritics such as Tabu Search or Simulated Annealing can help escape from local minima. Another alternative is to explore larger neighborhoods to improve the current best solution. By exploring larger neighborhoods, the need for metaheuristics becomes less important, as the search is less myopic. Building larger neighborhoods, however, often requires a great deal of expertise. A well-known and successful one for vehicle routing problems is the Lin-Kernighan neighborhood [LK73] that generalizes the 2-OPT move to K-Opt.

For some problems, exponentially sized neighborhoods can be explored in polynomial time using algorithms such as max-flow or path algorithms in graphs. We then speak of a very large-scale neighborhood search [Ahu+02].

These complex neighborhoods are often problem specific and difficult to adapt even for slight variations of the problem.

A more generic approach based on this idea of enlarging the neighborhood uses an optimization solver to explore the neighborhood [Sha98]. The main advantage is that the user expertise can be limited to the modeling of the problem rather than the design of complex move algorithms. This approach alternates between two phases: the random relaxation of a fraction of the decision variables, and the assignment of those variables using the solver as a black-box tool. This large neighborhood approach has been used with great success using constraint programming (CP) solvers to solve scheduling [Lab+18] or vehicle routing problems [JVH11]. Similar ideas have also been used with Mixed Integer Programming (MIP) solvers under the name of local branching [FL10].

Recently, some combinatorial optimization problems have been solved efficiently using DD approach [Ber+16b]. Generic solvers for these approaches have also been developed [GSC20a] using the techniques explained in Chapters 4 and 5. It is thus a natural idea to attempt exploring large neighborhoods using DD solvers.

6.2 Large Neighborhood Search

As explained above, LNS is an incomplete optimization method that aims at being able to escape local minima while freeing the practitioner from the need to devise complex, highly specialized metaheuristics. To this end, LNS attempts to find a balance between intensification (apply advanced inference algorithms to explore promising neighborhoods) and diversification (explore different neighborhoods). This is why starting from an initial solution s^*, LNS alternates between a relaxation phase and a reoptimization phase. During the relaxation phase, decisions made in s^* are challenged for a small fraction of the variables. Then, during the reoptimization, a solver operates a "black box" resolution of the remaining (sub-)problem. Whenever a solution s'^* improving the best known objective is discovered ($f(s'^*) < f(s^*)$); the incumbent best solution is updated.

Our Approach This thesis proposes to use restricted DDs as a means to explore sets of solutions in a large neighborhood of s^*. Algorithm 13 shows how this is done in practice. Similar to vanilla LNS, our method must strike a balance between intensification and diversification. In our case, the intensification target is achieved through the compilation of a restricted DD (Algorithm 13 lines 12-13).

In our method, three mechanisms are relevant to intensification. The first one is the (optional) use of an RLB procedure to discard nodes that cannot

Algorithm 13 LNS with Decision Diagrams

1: **Input:** a DP-model $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$
2: **Input:** $s^* \leftarrow$ the best known solution or none.
3: $MaxDepth \leftarrow |S| - 2$
4: $d \leftarrow MaxDepth$
5: **while** end criterion not met **do**
6: $\quad r' \leftarrow r$
7: $\quad$ **if** $s^* \neq none$ **then**
8: $\quad\quad$ // The next restricted DD that will be compiled will be rooted in
9: $\quad\quad$ // the dth node along the best known solution path
10: $\quad\quad r' \leftarrow s^*_d$
11: $\quad \mathcal{P}' \leftarrow \langle S, r', t, \bot, v(r'), \tau, h \rangle$
12: $\quad neigborhood \leftarrow CompileRestrictedDD(\mathcal{P}', s^*)$
13: $\quad neighbor \leftarrow ShortestPath(neigborhood)$
14: $\quad$ **if** $f(neighbor) < f(s^*)$ **then**
15: $\quad\quad$ // If the best solution that was found in the newly compiled neighborhood
16: $\quad\quad$ // improves the objective over the best known solution, then keep that new
17: $\quad\quad$ // solution and reset the depth d
18: $\quad\quad s^* \leftarrow neighbor$
19: $\quad\quad d \leftarrow MaxDepth$
20: $\quad$ **else if** $d = 0$ **then**
21: $\quad\quad d \leftarrow MaxDepth$
22: $\quad$ **else**
23: $\quad\quad d \leftarrow d - 1$

lead to an objective improvement. The second and third mechanisms follow from the behavior of the restriction procedure. As shown in Algorithm 14, one initially partitions the nodes of a given layer L_i between those nodes that *must* be kept in the layer, and the others (Algorithm 14 lines 5-11). This decision is based on the *mustKeep* predicate (Definition 6.2.1) which states that a node n from the ith layer must be kept if the value associated to variable x_i on the best $r-n$ path (denoted $p^*_{r-n}(i)$) is the same as the value of x_i in s^* (denoted $s^*(i)$). This guarantees that $s^* \in Sol(neighborhood)$ and hence that $neighborhood$ is an actual neighborhood of s^* (Algorithm 13 line 12).

Definition 6.2.1

$$mustKeep(n, s^*, i) \iff p^*_{r-n}(i) = s^*(i)$$

The last of our intensification mechanisms consists of the node-selection heuristic which is used to choose the nodes remaining in a layer after restriction. Algorithm 14 shows at lines 12-13 that the candidate nodes are filtered to keep only the best nodes according to their RLB.

There are two mechanisms at play in our method to ensure a fair amount of diversification. The first one consists of selecting a different root for the compilation of the restricted DDs. This is done in a systematic manner, optimistically starting with a node at the bottom of the DD which yielded the best solution; progressing towards the actual root of the problem (Algorithm 13 lines 6-11, 19-23). Doing so, our algorithm gets a chance to compile different DDs at each iteration, each of which generating exact or near-exact layers at different heights[1]. Which means that the different DDs that are compiled will likely offer a good view of the impact of perturbing sequences of decisions. This might be beneficial for scheduling problems – and might be challenged when solving a different kind of problem.

The second mechanism in use consists in the introduction of some randomness during a layer restriction. As shown in Algorithm 14, any node not satisfying the *mustKeep* predicate might still be forced into the restricted layer with a small probability (line 7).

Benefits Our approach offers several benefits. DDs leverage their underlying DP model as a means to explore the neighborhood of a given best solution. Moreover, as opposed to vanilla LNS, our approach is sometimes able to *prove*

[1]Because the considered DDs are compiled top-down, chances are that even though the compiled DD is *restricted* – that is, an under approximation of the true *exact DD*, the first few layers of that diagram will be complete or nearly complete. This means the compiled DD is less “myopic” near its top than its bottom, and consequently offers a view of the impact of perturbing a given portion of the solution (the information is more accurate concerning the decisions made at the top of the restricted DD).

Algorithm 14 Restrict Procedure

```
1: Input: L_i : the layer that needs to be restricted
2: Input: s* : the best known solution, or none
3: Input: W : maximum layer width
4: Input: p : a small probability (e.g. 10%)
5: frontier ← 0
6: for k ∈ {0..|L_i|} do
7:    if mustKeep(L_i[k], s*, i) ∨ random() ≤ p then
8:       swap(L_i, frontier, k)
9:       frontier ← 1 + frontier
10: keep ← nodes[0..frontier[
11: candidates ← nodes[frontier..|nodes|[
12: sort(candidates based on their RLB)
13: truncate(candidates, max(0, W − |keep|)
14: L_i ← concat(keep, candidates)
```

the optimality of the instances it solves. This is usually only possible when using an exact method such as MIP or Branch-and-Bound [Ber+16b]. Indeed, our method proceeds by generating sets of complete solutions at each iteration. Still, because the value of the best incumbent solution is improving over time, so is the pruning power of the RLB used when compiling the DD. From there, it follows that sometimes the pruning power of RLB is sufficient to let the DD compile without requiring any restriction ($d = MaxDepth$ and no layer ever exceeds the maximum width). In that event, the resulting DD is an exact DD which proves the optimality of the best solution it contains. Naturally, this capability stems from a tradeoff between the pruning power of RLB and the maximum layer width of the DD that are used. Therefore, the algorithm will not be able to always achieve a formal proof of optimality. Predicting whether it will succeed in delivering such a proof is undecidable. Still, we believe that the possibility for a metaheuristic approach to sometimes give a proof of optimality is an appreciable feature.

Another benefit of using our method comes from the configurable aspect of the DD compilation. Which means one can choose how wide the DD is allowed to be. And therefore how long it will take to compile the DD[2]. Thereby arbitraging a balance between diversification and intensification.

[2] Incidentally, the likelihood that DD be exact

6.3 Experimental Study

In order to evaluate the effectiveness of DD-LNS, we considered two sequencing problems: the pigment sequencing problem (PSP) and the traveling salesman problem with time windows (TSPTW), both of which have been presented in Chapter 3. The source code of all models and benchmark instances are publicly available online at `https://github.com/xgillard/ijcai_22_DDLNS`.

All our experiments were performed on the same physical machine equipped with two Intel(R) Xeon(R) CPU E5-2687W v3 and 128Gb of RAM. On that machine, each considered solver was given a 10 minutes timespan using a single thread and a maximum memory quota of 2Gb in order to solve each instance.

PSP We used our own generic DD-LNS implementation in which we plugged the DP model given above and compared its performance against the state-of-the-art MIP model (PIG-A-3) from [PW06]. Because of the simplicity of our DP model, and because PIG-A-3 was tuned by MIP experts over a decade; we also included the simpler MIP models PIG-A-1 and PIG-A-2 in our comparison. These should give an idea of what a practitioner might reasonably expect when creating a model for the PSP. All MIP models originate from [PW06] and are written using FICO Xpress Mosel v8.11. Our experiments bear on the 500 instances from the second set of benchmarks that have been presented in Chapter 3 (the instances with $|\mathcal{I}| = 10$). This set of benchmark has been specifically chosen because none of these instances could be solved by the models from previous chapters, either with pure dynamic programming or with branch-and-bound mdd [Ber+16b; Gil+21].

TSPTW The experiments involve the same generic DD-LNS framework used for PSP; using the DP model presented above. We compared its performance vs a CP model implemented in Choco 4.10.6 [3] using a LNS that re-optimize a small sequence[4] of 5 decision variables $x_i \dots x_{i+4}$ randomly selected at each restart. The restart strategy that has been used triggers a restart after the 30th failed attempt to improve the current best solution.

Our experiments bear on the 467 instances of the benchmark suites which are usually used to assess the efficiency of new TSPTW solvers.

Results Table 6.1 shows the number of PSP instances for which the best solution found by each solver matches the best known solution. It also shows

[3] https://choco-solver.org/

[4] Several alternative relaxation schemes and parameters were experimented and this one gave the best results.

Method	W	Best Known	1% Gap
PIG-A-1	N.A.	25	232
PIG-A-2	N.A.	86	226
PIG-A-3	N.A.	473	499
DD-LNS	10	167	449
DD-LNS	100	276	489
DD-LNS	1000	368	490

Table 6.1: Number of PSP instances for which the optimum solution has been found and for which the best solution found is within 1% of the best known solution.

the number of instances where the best solution found was within 1% of the globally best known solution ($\frac{found - best_known}{best_known} \leq 1\%$). From this table, it clearly appears that the combination of LNS with Decision Diagrams is very efficient at finding good solutions. Indeed, this method outperforms the PIG-A-1 and PIG-A-2 models in all situations; even with a maximum layer width as small as 10 nodes. Furthermore, in spite of the simplicity of its underlying DP model, our DD-LNS approach fares comparably to the much more advanced PIG-A-3 models.

Because the TSPTW satisfiability is NP-complete, and in order to establish a fair comparison between CP-LNS and DD-LNS, we bootstrapped the problem resolution of all solvers with an initial feasible solution computed off-line (the same for both CP and DD-LNS). These initial solutions have been computed by a variant of [DSU10]. Table 6.2 shows the number of TSPTW instances for which the best solution found by each solver matches the published best known solution. It also shows the number of instances where the best solution found was within 1 % of the overall best known solution. This table shows that both methods are highly efficient at finding good solutions to the TSPTW; DD-LNS having a slight edge over CP-LNS. During this phase of the experiments, we identified 75 new solutions with an objective value matching that of the published best known solution.

As a mean to assess the effectiveness of these methods at optimizing TSPTW independently of the initial solution, we repeated the experiment; this time initializing the resolution with the published best known solution of each instance. This again proved the high efficiency of both methods. Both methods identified new solutions improving the objective value of standard benchmark instances. In practice, DD-LNS was able to identify 8 improving solutions in two benchmarks suites (AFG and OhlmannThomas) and CP-LNS was able to find 10 new solutions in the OhlmannThomas benchmarks suite.

An interesting general observation to make about our experiments stems from the fact that the maximum width W of the compiled DDs provides an easy means to tune the diversification level. Indeed, both Table 6.1 and Ta-

Method	W	Best Known	1% Gap
CP-LNS	N.A.	144	184
DD-LNS	10	197	246
DD-LNS	100	217	261
DD-LNS	1000	217	249

Table 6.2: Number of TSPTW instances for which the optimum solution has been found and for which the best solution found is within 1% of the overall best known value

ble 6.2 show that increasing the maximum layer width W improves the solver performance before it starts hampering it.

6.4 Related Work

A recent line of work [RCR18], might seem similar to ours at first glance. Indeed, both method blend Decision Diagrams with local search metaheuristics. However, their approach is fundamentally different from ours: the point of their method is to use local search as a means to *compile* relaxed DDs in order to compute bounds on the optimal problem value. The bounds that are derived this way are very tight, but feasible solutions cannot be extracted from the DD. Our method, on the other hand, uses *restricted* DDs as a means to automatically generate good quality neighborhoods comprising only feasible solutions.

Our approach can be considered as a hybridization of DD for optimization (DDO), beam search, LNS, and the phase saving heuristic which is customarily used in SAT solvers [PD07]. Combinations of some of these ingredients have recently been proposed, but, to the best of our knowledge, no approach ever combined all of them. For instance, [LIB10] hybridized beam search with Ant-Colony Optimization (ACO) in order to solve the TSPTW. As opposed to our method, it is driven by an ACO component rather than DP.

The very recent [DCS18] and [Bjö+20] pursue a goal similar to ours. They try to blend constraint programming, phase saving and LNS to solve hard combinatorial problems but focusing on propagators while this work relies on a dynamic programming formulation of the problem only.

6.5 Conclusions

We introduced and evaluated a method combining large neighborhood search with decision diagrams to solve hard combinatorial optimization problems having a dynamic programming formulation. Its simplicity and good performances (experimented on two problems) might be appreciable for practitioners. In particular, when one has to repeatedly take good decisions quickly; as

would for instance be the case when adapting a production schedule based on an ever-evolving order book.

A Global Minded Restricted DD Compilation Method

7

Contributions and Publication Information

At the time of writing, the ideas presented in this chapter have not been published yet. The contribution of this chapter consists of a two-step algorithm inspired by approximate dynamic programming that uses DD to both derive good bounds on the objective value, and to find an actual feasible solution to the optimization problem.
While the technique presented in this chapter is generic and applicable to a broad range of problems, no experiment has been conducted (yet) to assess its effectiveness on other problems than the Pigment Sequencing Problem (PSP). Which is why the coming pages are heavily PSP-oriented. The application of our method to different problems is – at the time being – still future work.

In Chapter 5 we proposed a rough lower bound as a way to improve the quality of the lower bound derived from the relaxed MDDs. This, however, is not the only bound that can be tightened using MDDs. This chapter presents a novel approach to compiling restricted MDDs which helps to tighten the upper bounds derived from these DDs. As opposed to the classical restricted MDD compilation scheme, our strategy uses a two-phased approach which is 'global minded' contrary to the usual *MinLP*[Ber+14b] heuristic consisting in a blind layer-by-layer node suppression. Which is why, our alternate strategy is able to swiftly produce restricted MDDs encoding very good feasible solutions.

7.1 Motivation

As shown in the previous chapters, optimizations techniques based on dynamic programming and decision diagrams can prove highly effective. In some cases, however, the state spaces of the DP models are simply too large and the bounds derived from restricted and relaxed MDD are of little to no

use. This happens, for instance, when the local-minded "MinLP" heuristic provokes the deletion of all nodes that might possibly lead to a feasible solution. In such a 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 shrink the optimality gap. In the absence of a *perfect heuristic*[1], this situation is unavoidable and bound to occur at least in some circumstances. Still, it motivated our search for a better – global minded – approach that could increase the usefulness of the compiled restricted DDs.

7.2 Intuition

The technique we propose builds on the observation that for most combinatorial problems, the small instances are often easier to solve than large ones. Obviously this statement is not true of *all* instances and all problems[2]. Still, it is true that whenever a combinatorial problem is small enough, it can easily be solved to optimality with a bruteforce approach. This was the ground intuition behind research on approximate dynamic programming which took place throughout the 1980's and 1990's [Zip80; BBS87; Rog+91] and it is at the heart of our technique as well.

Based on the above observation, we propose to proceed in 1+3 steps as shown per figure 7.1. Before solving the problem instance $\mathcal{P}$, we propose to preprocess it to create $\mathcal{P}'$, a new (derived) instance whose size is much smaller than the original one and can be seen as an over approximation of $\mathcal{P}$. Then, whenever a restricted DD needs to be compiled to explore the portion of the state space behind a node u, a reduction is performed to bring u from $\mathcal{P}$ to $\mathcal{P}'$ and the subproblem $\mathcal{P}'_u$ (the fragment of $\mathcal{P}'$ which is rooted in u) is solved to optimality. If the preprocessing of $\mathcal{P}$ has created a small enough $\mathcal{P}'$, chances are high that solving $\mathcal{P}'_u$ will be an easy task. Then, the best solution to $\mathcal{P}'_u$ is used as an heuristic to guide the compilation of a restricted DD for $\mathcal{P}_u$.

There are numerous ways in which this general framework can be adapted to fit to a given problem. For instance, it does not prescribe how an instance must be preprocessed to yield a much smaller instance, which leaves plenty of room for further investigation. A few investigation leads could be:

- considering aggregates as "virtual items" that must be instantiated when compiling the restricted DD;

[1] A *perfect heuristic* is an oracle that is never wrong and always capable of suggesting the best choice.

[2] For instance, the boolean satisfiability problem (SAT) exhibits a "phase transition" behavior [GW94]. This means that instances on either side of the phase transition are relatively easy to solve, even the large ones. However, even the small instances on that phase transition can be very hard to solve.

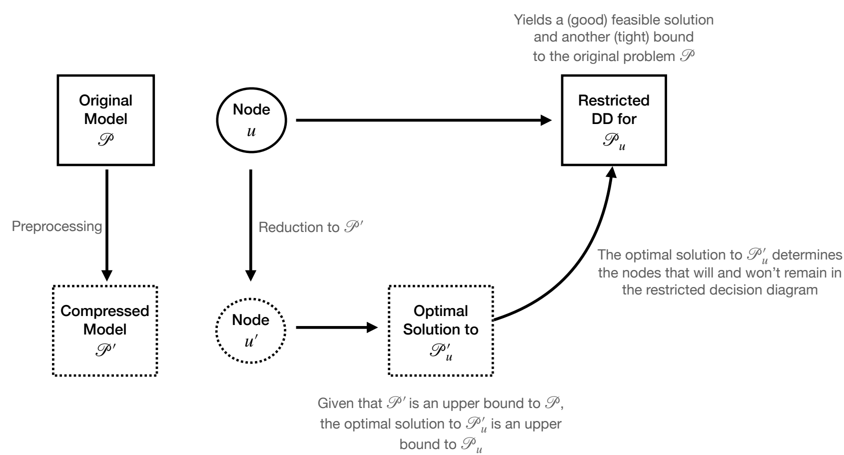


Figure 7.1: Intuition behind the technique. During a phase of preprocessing, a much simpler problem $\mathcal{P}'$ is derived from the original instance $\mathcal{P}$. The instance $\mathcal{P}'$ is as an approximation of $\mathcal{P}$. Whenever a restricted DD needs to be compiled for some node u during the branch-and-bound; an equivalent is computed for u in $\mathcal{P}'$. This reduction enables the computation of the exact optimum to $\mathcal{P}'_u$ (hence an approximation of $\mathcal{P}_u$). That exact solution is then used as an heuristic to guide the compilation of the restricted DD.

- considering only the possible decisions within an aggregate until it is exhausted before to start considering other decisions;
- assigning the same value to all variables within an aggregate,
- ...

All these options are obviously problem dependent and problem specific. They seem to be interesting nonetheless as they could be used to guide a branch-and-bound with DD solver towards the optimal solution using problem specific knowledge.

7.2.1 Visual Example

Before we delve into the specifics of a detailed complete example, let us consider the simple example from figure 7.2 which visually conveys the intuition given above. It shows how a TSP instance could be solved using our method. The original problem $\mathcal{P}$ comprises a large number of cities which makes the compilation of a *good* restricted DD harder than it should (7.2-a). During a preprocessing phase, the problem instance has been compressed so as to create an easier problem $\mathcal{P}'$. Whenever a restricted DD must be compiled to explore the solution space under a node u, that node is converted into an equivalent node of $\mathcal{P}'$ which is then solved to optimality very easily (7.2-b). The optimal solution is then used as a means to guide the compilation of a good restricted DD (7.2-c). In the case of the considered example, the heuristic would then consist in considering the subtours in the aggregate and then concatenating them according to the order of the optimal solution to the compressed problem $\mathcal{P}'$. Doing so, the restricted DD develops complete tours that are feasible solutions of $\mathcal{P}$.

7.3 Detailed Example

In order to further clarify the various steps or our approach and to explain the details of the experimental study we have conducted, we will use a PSP instance $\mathcal{P}$ formulated as explained in Chapter 3. The details of the instance $\mathcal{P}$ are given below, and will serve as a running example throughout the rest of this chapter. This instance describes a planning problem with an horizon $\mathcal{H}$ of five time steps. Over the course of that period, three kinds of items have to be produced $\mathcal{I} = \{0, 1, 2\}$. The stocking cost of each item is given with the $\mathcal{S}$ vector. Thus, stocking one unit of item 0 for one period of time ($\mathcal{S}_0$) costs 20. Similarly, the stocking of one unit of item 1 costs $\mathcal{S}_1 = 10$ and one unit of 2 costs $\mathcal{S}_2 = 5$. The changeover matrix C indicates the cost of changing the machine configuration from producing one item to any other.

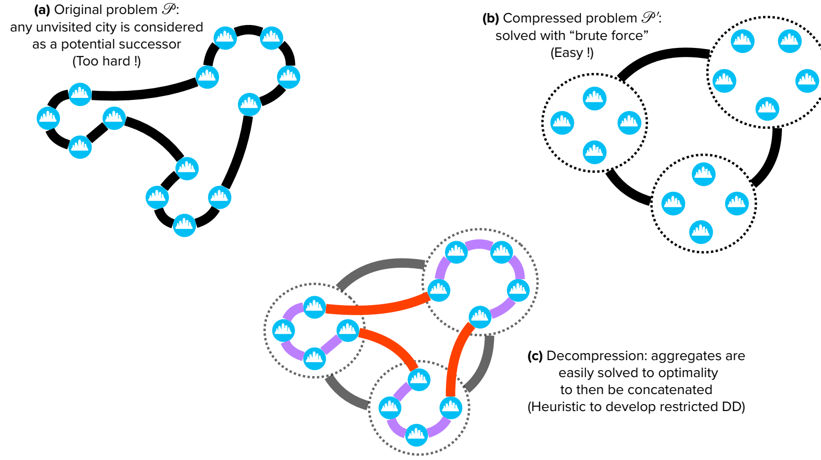


Figure 7.2: Visual example to convey the intuition of the method: a hard TSP problem (a) is reduced to a much simpler one (b) whose optimal solution serves as an heuristic to guide the compilation of restricted DD (c).

For instance, changing the configuration from the production of item 0 to 1 costs $C_{0,1} = 1$. Similarly, changing the configuration from 2 to 0 incurs a cost $C_{2,0} = 4$. Finally, the vector Q_0 of demands for item 0 tells us that an item of that type must be delivered to some customer at time 3 and 4. Q_1 tells us that two units of item 1 must be produced, and these must be delivered at time 2 and three. Finally, Q_2 shows that only one unit of 2 must be produced, and it must be delivered at time 1.

$$\begin{array}{ll|ll|ll} \mathcal{H} & = 5 & & \begin{pmatrix} 0 & 1 & 2 \\ 3 & 0 & 2 \\ 4 & 1 & 0 \end{pmatrix} & Q_0 & = (0,0,0,1,1) \\ \mathcal{I} & = \{0,1,2\} & C = & & Q_1 & = (0,0,1,1,0) \\ \mathcal{S} & = (20,10,5) & & & Q_2 & = (0,1,0,0,0) \end{array}$$

Based on that data, the previous demands $\mathcal{T}$ are derived as shown below:

Item 0	$\mathcal{T}_0^0 = -1$	$\mathcal{T}_1^0 = -1$	$\mathcal{T}_2^0 = -1$	$\mathcal{T}_3^0 = -1$	$\mathcal{T}_4^0 = 3$	$\mathcal{T}_5^0 = 4$
Item 1	$\mathcal{T}_0^1 = -1$	$\mathcal{T}_1^1 = -1$	$\mathcal{T}_2^1 = -1$	$\mathcal{T}_3^1 = 2$	$\mathcal{T}_4^1 = 3$	$\mathcal{T}_5^1 = 3$
Item 2	$\mathcal{T}_0^2 = -1$	$\mathcal{T}_1^2 = -1$	$\mathcal{T}_2^2 = 1$	$\mathcal{T}_3^2 = 1$	$\mathcal{T}_4^2 = 1$	$\mathcal{T}_5^2 = 1$

In line with the DP model given in Chapter 3, Figure 7.3 illustrates what an exact MDD compiled for the problem $\mathcal{P}$ would look like. In this picture, the decision label are shown above the layer separation line (red), and the weights (transition cost) of the arcs are written below it (green). The shortest path in this DD is boldfaced. From this picture, it is thus easy to see that the optimal solution to this example is $[\![x_0 = 2, x_1 = 1, x_2 = 1, x_4 = 0, x_4 = 0]\!]$ and its value is 29.

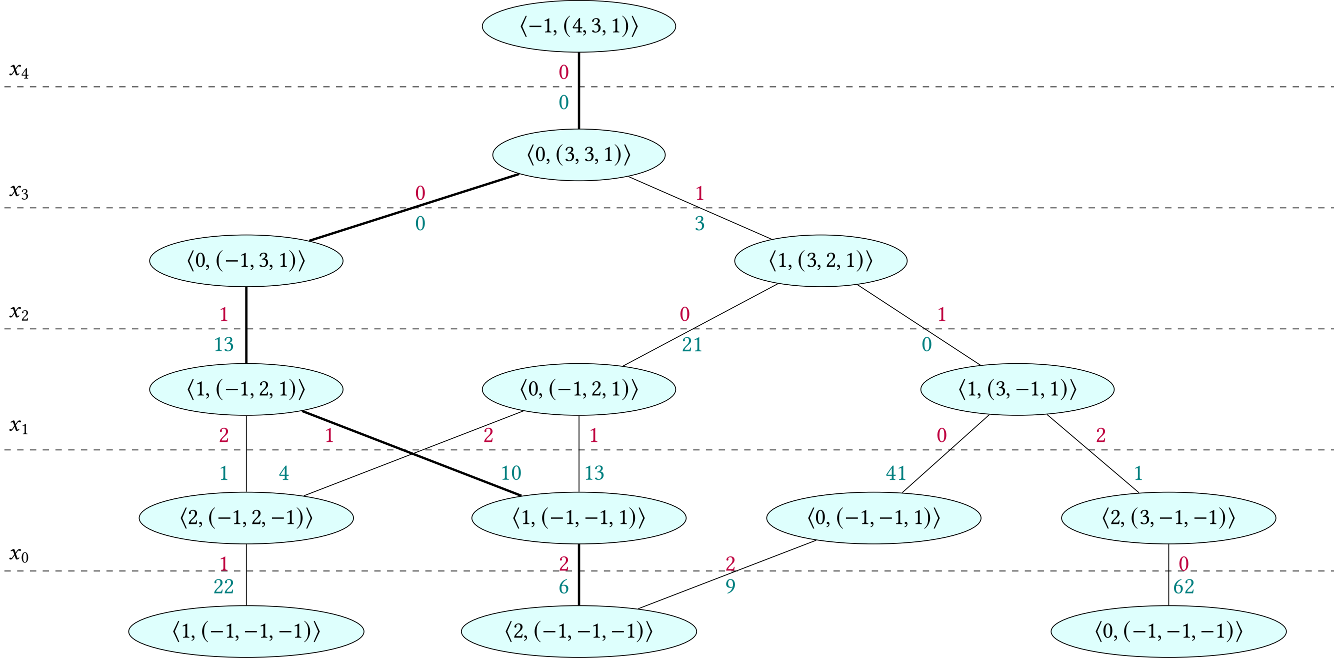


Figure 7.3: Visual representation of an exact MDD compiled from the example instance $\mathcal{P}$. The decision labels of the arcs are depicted above the layer separation lines. The weights (transition cost) of the arcs are written below it and the shortest path is boldfaced.

7.4 Preprocessing: Compressing the instance

As explained above, our method requires a phase of preprocessing at its start. During that phase our method *compresses* the PSP instance $\mathcal{P}$ into an easier instance $\mathcal{P}'$ whose number of items is arbitrarily chosen. In $\mathcal{P}'$, each item stands for a class of items that are similar to one another in $\mathcal{P}$.

7.4.1 Partitioning the original $\mathcal{I}$ in classes of similar items

In order to partition the original item set $\mathcal{I}$ in w distinct classes of similar items, we use a k-means algorithm which is initialized with the k-means++ method [CKV13; AV06]. To that end, we consider each item $i \in I$ as a coordinate $(i_0, i_1, \ldots, i_n)$ from an n+1 dimensional system. The first n members i_j $(0 \leq j < n)$ of the coordinate simply correspond to the changeover costs $C_{i,j}$ between the items i and j in $\mathcal{P}$. The last member i_n of the coordinate corresponds to the stocking cost $\mathcal{S}_i$ of item i in $\mathcal{P}$. This representation lets us compute the distance between any two items as the generalized Euclidean distance between their coordinates[Tab14].

$$d(x, y) = \sqrt{\sum_{0 \leq i < n} (x_i - y_i)^2}$$

Example Assuming that we would like to rewrite our running example from section 7.3 so as to create a simpler instance $\mathcal{P}'$ having only two items instead of three, we would have the following coordinates for items 0, 1 and 2:

$$\begin{aligned} Item\ 0 &= (0, 1, 2, 20) \\ Item\ 1 &= (3, 0, 3, 10) \\ Item\ 2 &= (4, 1, 0, 5) \end{aligned}$$

The "distances" between these items are thus respectively:

$$\begin{aligned} dist(Item\ 0, Item\ 1) &= \sqrt{(0-3)^2 + (1-0)^2 + (2-3)^2 + (20-10)^2} &= \sqrt{111} \\ dist(Item\ 0, Item\ 2) &= \sqrt{(0-4)^2 + (1-1)^2 + (2-0)^2 + (20-5)^2} &= \sqrt{245} \\ dist(Item\ 1, Item\ 2) &= \sqrt{(3-4)^2 + (0-1)^2 + (3-0)^2 + (10-5)^2} &= \sqrt{36} \end{aligned}$$

Therefore, the best possible partitioning returned by the k-means algorithm would be: $Clus_0 = \{0\}, Clus_1 = \{1, 2\}$.

7.4.2 Rewriting a compressed instance

Knowing the original PSP instance $\mathcal{P}$, and based on the partitioning $Clus_0$, $Clus_1, \ldots, Clus_{w-1}$ obtained from the k-means clustering, we can compile the compressed instance $\mathcal{P}'$ as follows. The set of items $\mathcal{I}'$ simply comprises

the arbitrary w classes from above. Also, the time horizon of the instance remains unchanged. That is, we have $H' = H$. The stocking cost of each of the w items from the compressed instance actually correspond to the stocking cost dimension of the centroid from each cluster. In other word, the stocking cost of the ith item of the compressed instance is given by

$$\mathcal{S}'_i = \text{avg}\left\{\mathcal{S}_j \mid j \in Clus_i\right\} \ \textbf{for}\ 0 \leq i < w$$

Similarly, the changeover cost between two items $i, j \in \mathcal{I}'$ of the compressed instance is computed as the average changeover cost between any two items from the sets $Clus_i$ and $Clus_j$. This means that the compressed changeover cost matrix C' is a square $w \times w$ matrix where

$$C'_{i,j} = \text{avg}\left\{C_{a,b} \mid a \in Clus_i, b \in Clus_j\right\} \ \textbf{with}\ i, j \in \mathcal{I}'$$

Obviously, the demands for each of the w compressed items amounts to the sum of the demands for the items which they stand for. Because of the normalization assumption, the vector $\mathcal{Q}'$ of demands must be rebuilt using a very simple algorithm. First, a denormalized version of $\mathcal{Q}'$ is computed s.t. $\mathcal{Q}_i'^t = \sum_{a \in Clus_i} \mathcal{Q}_a^t$ for each time step t and item i of the compressed instance. Then, these quantities are normalized. A backwards traversal starting at the time horizon spreads the demands $\mathcal{Q}_i'^t > 1$ onto earlier times; thereby ensuring that $\mathcal{Q}_i'^t \in \{0, 1\}$ for all times t and items i.

Example Building on the clustering obtained in example 7.4.1, the instance $\mathcal{P}'$ would be rewritten:

$$\begin{aligned} \mathcal{H}' &= 5 \\ \mathcal{I}' &= \{0, 2\} \\ \mathcal{S}' &= (20, 7.5) \end{aligned} \quad \Bigg| \quad C' = \begin{pmatrix} 0 & 1.5 \\ 3.5 & 0 \end{pmatrix} \quad \Bigg| \quad \begin{aligned} \mathcal{Q}'_0 &= (0, 0, 0, 1, 1) \\ \mathcal{Q}'_1 &= (0, 1, 1, 1, 0) \end{aligned}$$

And from that rewriting, the previous demands for each clustered item of the adapted problem are derived as follows:

Item 0'	$\mathcal{T}_0'^0 = -1$	$\mathcal{T}_1'^0 = -1$	$\mathcal{T}_2'^0 = -1$	$\mathcal{T}_3'^0 = -1$	$\mathcal{T}_4'^0 = 3$	$\mathcal{T}_5'^0 = 4$
Item 1'	$\mathcal{T}_0'^1 = -1$	$\mathcal{T}_1'^1 = -1$	$\mathcal{T}_2'^1 = 1$	$\mathcal{T}_3'^1 = 2$	$\mathcal{T}_4'^1 = 3$	$\mathcal{T}_5'^1 = 3$

7.5 During Resolution: Better Upper Bounds

As outlined in the introduction to this chapter, whenever an upper bound needs to be derived from a state σ_t, we apply a two phase process. First, σ_t is transformed into a corresponding state σ'_t from the compressed instance and the subproblem described by σ'_t is solved to optimality. In a second time, the optimal solution to the compressed subproblem σ'_t is used to compile an actual restricted MDD which heuristically yields stronger upper bounds.

7.5.1 Phase 1: Compressing σ_t as σ'_t and solving it

Intuitively, the compression of a state resembles the fusion operated during relaxation with the $\oplus$ operator. The difference being that instead of combining items from different nodes disagreeing on the remaining quantities, items are combined according to the preprocessed clustering.

In order to show how the "state compression" is done in practice, it helps to start defining the *remains* function. This function tells for each item $i \in \mathcal{I}$ and state $\sigma = \langle k, u \rangle$ the quantity of item i that still needs to be scheduled in the subproblem rooted in σ. Formally, it is defined as follows:

$$remaining(\langle k, u \rangle, i) = \left| \left\{ \mathcal{T}_j^i \mid 0 \le j \le \mathcal{H}, 0 \le \mathcal{T}_j^i \le u_i \right\} \right|$$

Example Based on our running example, we can use the *remaining* function to determine that state $\sigma = \langle 0, (3,3,1) \rangle$ corresponds to a situation where one unit of item 0, two units of item 1 and one unit of item 2 are still to be scheduled. Indeed we have:

$$\begin{aligned} remaining(\sigma, 0) &= \left| \left\{ \mathcal{T}_4^0 \right\} \right| \\ &= |\{3\}| \\ &= 1 \end{aligned}$$

$$\begin{aligned} remaining(\sigma, 1) &= \left| \left\{ \mathcal{T}_3^0, \mathcal{T}_4^0, \mathcal{T}_5^0 \right\} \right| \\ &= |\{3,3,2\}| \\ &= |\{3,2\}| \\ &= 2 \end{aligned}$$

$$\begin{aligned} remaining(\sigma, 2) &= \left| \left\{ \mathcal{T}_2^2, \mathcal{T}_3^2, \mathcal{T}_4^2, \mathcal{T}_5^2 \right\} \right| \\ &= |\{1,1,1,1\}| \\ &= |\{1\}| \\ &= 1 \end{aligned}$$

On that basis, the state $\sigma' = \langle k', u' \rangle$ is computed from $\sigma = \langle k, u \rangle$ as

$$\begin{aligned} k' &= j \ \textbf{ s.t. } \ k \in Clus_j \\ u'_i &= \mathcal{T'}^i_{\sum\limits_{j \in Clus_i} remaining(\sigma, j)} \qquad \textbf{for all } 0 \le i < \mathcal{H} \end{aligned}$$

Intuitively, this state compression means that the previous item k of the original state is simply mapped onto the equivalence class (the cluster) which stands for it in the simpler problem. Likewise, the previous demands of each clustered item consider the total number of goods that must be still scheduled in the original problem for any of the items belonging to that cluster.

Example Continuing our running example, the state $\sigma = \langle 0, (3, 3, 1) \rangle$ would be compressed as $\sigma' = \langle 0, (3, 3) \rangle$.

7.5.2 Phase 2: Inflating the optimal compressed solution

In order to find a good upper bound (that is a good feasible solution) for the subproblem σ_t based on the optimal solution of σ'_t, one "inflates" it into a restricted MDD. In practice, this is achieved by repeatedly expanding the transition relation (as in the usual case). However, this expansion is not done for every possible transition. Instead, only the paths that might correspond to the optimal solution of σ' are explored.

Example Applying the node compression from previous section to the root node r, would yield a compressed root node $r' = \langle -1, (4, 3) \rangle$. The optimal solution to $\mathcal{P}'$ rooted in r' (that is, the optimal to $\mathcal{P}'$ as a whole) is $[\![x'_0 = 1, x'_1 = 1, x'_2 = 1, x'_3 = 0, x'_4 = 0]\!]$. And the objective value of that solution is 26. Based on that solution, one can compile the restricted MDD shown in Figure 7.4. This MDD is computed "inflating" the compressed solution into actual feasible solutions. The length of the shortest path in that MDD (and hence, the lower bound obtained from this DD) is 29; which is also the optimal value of $\mathcal{P}$.

As can be seen from Figure 7.4, not all paths are expanded while inflating a compressed solution. For instance, items 1 and 2 were not even considered when deciding on the value attributed to variable x_3 and x_4. This is because the corresponding decisions made in the compressed solutions were $[\![x'_3 = 0, x'_4 = 0]\!]$ but $1, 2 \notin Clus_0$. The rest of the MDD is developed as per Chapter 2.

7.6 Evaluation

In order to evaluate the effectiveness of the technique presented in this chapter, we performed a computational experiment using the same PSP benchmark instances that have been presented in Chapter 3. We attempted to solve each of these instances using a variation of the framework presented in Chapter 4, adapting it to cope with the two phase "compression"-"inflation" process we just introduced. In our experiments, the approximate DDs were allowed to grow up to a maximum width of 10 nodes per layer. The implementation of our "compression" algorithm used $w = \left\lceil \frac{|\mathcal{I}|}{2} \right\rceil$. All our experiments were performed on the same physical machine equipped with an Intel(R) Xeon(R) CPU E5-2687W v3 and 128Gb of RAM. On that machine, each solver was allotted a 10 minutes timespan using one thread and a maximum memory footprint of 3Gb to solve each instance.

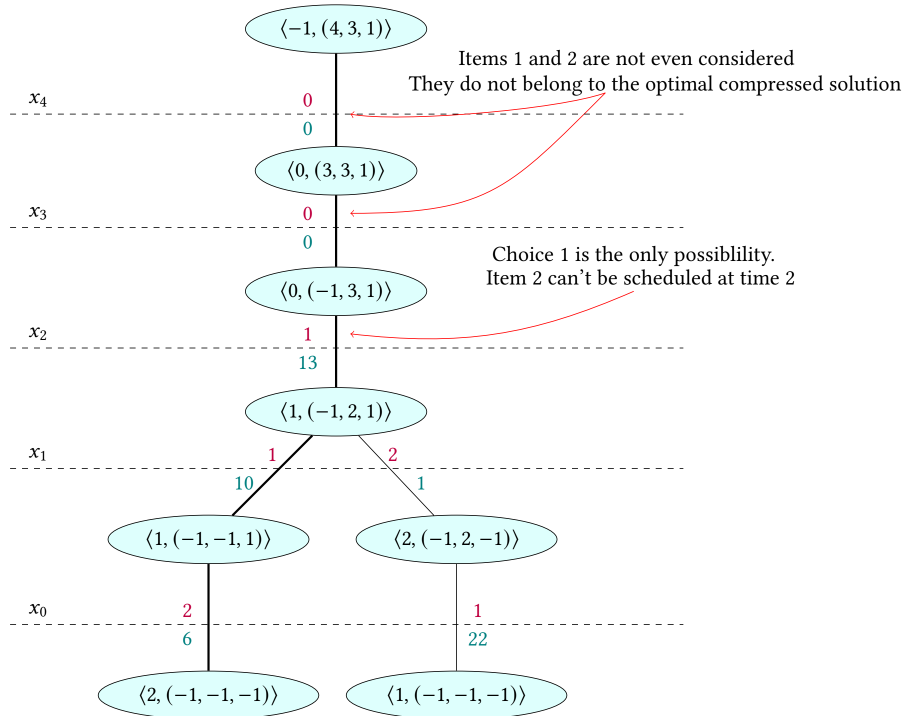


Figure 7.4: Restricted MDD derived from "inflating" the optimal compressed solution $[\![x'_0 = 1, x'_1 = 1, x'_2 = 1, x'_3 = 0, x'_4 = 0]\!]$. The decision labels of the arcs are depicted above the layer separation lines. The weights (transition cost) of the arcs are written below it and the shortest path is boldfaced.

The main result from this experimental study is shown in Figure 7.5 which compares the quality of the best solution found by a DDO+RUB solver (Figure 7.5 (a)) and that of a solver using our alternative restricted DD compilation scheme (Figure 7.5 (b)) with the globally best known solution. On these plots, a mark on the diagonal indicates that the best known solution has been found, and a mark above that line indicates a lower quality solution. The farther away from the diagonal, the larger the deviation from the best solution and hence the lower the quality of the best solution found by a given method.

Comparing these two plots, it clearly appears that marks on the right hand graph form a tighter group and are in general closer to the diagonal. This suggests solutions found using the compress-inflate compilation scheme for restricted DD tend to be of better quality than those found using the "vanilla" MDD approach that has been presented in previous chapters. This observation is confirmed by a numeric analysis of these results. Table 7.1 indeed shows that the gap to best known solution – that is the normalized distance between the best solution found with a given method and the best known

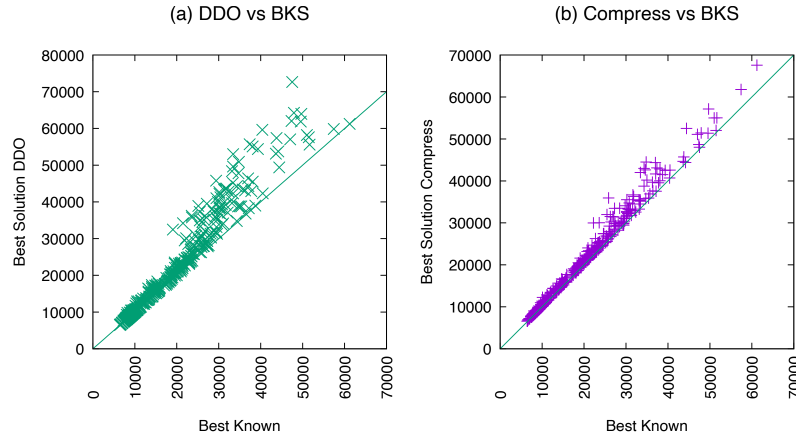


Figure 7.5: Comparison of the quality of best solution found

solution (BKS) $\frac{solution-BKS}{BKS}$ – is in general much narrower when using our two phased algorithm. The contrast is particularly stark when considering the hardest set of benchmark instances (instances having 10 items).

7.7 Limitations

It must, however, be noted that the quality of the solutions found with this new algorithm is not *as good* as the quality of the solutions obtained using the local search approach from Chapter 6. Indeed, the average gap with that method would be 0.06% with that method and the standard deviation 0.2%.

It is also worth mentioning that in absolute terms, the plain DDO+RUB method found solutions that are closer to the BKS more often than the compress-decompress approach (500 instances). This concern should, however, be nuanced as most of these occurrences happened in the simpler set of benchmark intances (instances having 5 items). Given that our experiments used $w = \left\lceil \frac{|\mathcal{I}|}{2} \right\rceil$, the compressed version of these instances only comprised three items which might have been too coarse of a relaxation. This is why we consider these results as promising regardless of the above limitations.

Average Gap to Best Known Solution		
	All Instances	*Hardest Instances*
Compress	**3.0%**	**4.9%**
DDO+RUB	4.5%	9.2%
Std Dev. in Gap to Best Known Solution		
	All Instances	*Hardest Instances*
Compress	**4.5%**	**5.0%**
DDO+RUB	9.6%	12.9%

Table 7.1: Gap to Best Known Solution (BKS)

7.8 Conclusions and further extensions

This chapter presented an alternate, global minded restricted-DD compilation scheme in two phases where a PSP (sub-)problem is explored by first solving an approximation of the original problem and then developing the solution to that approximation into a set of actual solutions. Then, an experimental study showed that the quality of the solutions found for the hardest PSP instances was improved by using this novel compilation scheme. While the results of this experimental study might not be as impressive as the ones from the LNS approach from the previous chapter, these results are deemed promising nonetheless. Still, this work could be extended in many directions: first, it might be interesting to devise nice abstractions over the compression-decompression mechanisms so as to turn this approach into a general-purpose technique. It might also be interesting to hybridize this line of work with the LNS method that was presented in the previous chapter. In that case, this alternate compilation scheme might for instance serve to find a good quality initial solution. In that context, it might for instance be used to increase the diversification during LNS reoptimization, e.g. by using the solution to the compressed problem when defining the $mustKeep$ predicate.

Another possible extension of this preliminary work would consist in searching for a similar global minded compilation scheme for relaxed DD. This could have the potential to deliver much tighter lower bounds on the optimal solutions and hence to speed up the problem resolution as a whole.

Conclusion | 8

The goal of this thesis was to deepen our knowledge of decision diagrams as tools to solve discrete combinatorial optimization problems. To start our uncovering of the topic, Chapter 4 investigated the engineering challenges imposed by the implementation of a generic library to build fast and efficient DD based optimization solvers. As a motivation, this chapter started with a brief presentation of the *ddo* library and an extensive example of how this library might be used to implement a solver for the knapsack problem. Then, it pursued with more implementation-related concerns which arise because of the CPU-and-memory intensiveness of DD-based solvers. Specifically, it addressed the problem of finding an efficient implementation for the DD abstraction. Four possible representations of the same mathematical objects have been proposed, and – as we have seen – the `VectorBased` architecture came out on top as it was able to leverage pre-allocation to achieve good cache locality and hence faster compilation times.

Our quest to improve the performance of DD based solver continued in Chapter 5 where we presented and evaluated the impact of two reasoning rules to strengthen the filtering of these solvers (LocB and RUB). Both techniques attach node-specific bounds to the nodes composing a DD but they differ in how these bounds are computed and when they are used. The LocB technique exploits the structure of the compiled DDs to derive these bounds whereas RUB relies on a fast-to-compute problem-specific bounding procedure for its purpose. Both of these techniques have been evaluated in an experiment bearing on the MISP, MCP, MAX2SAT and TSPTW. In particular, this experimental study showed the worthiness of devising simple problem specific bounding procedures. It also showed that exploiting the structure of the DD to strengthen the bounds that are derived from these DDs might help a great deal improving the solver performance. Furthermore, that same computational study revealed that our new rules complete one another. Which is why, a combination of both RUB and LocB supersedes the benefit of any single reasoning technique.

The next chapter investigated the hybridization of the DD-based solver technology with a Large Neighborhood Search paradigm. Doing so, it abandoned the guarantee of always being able to prove the optimality of a solution in exchange for being able to find good feasible solutions to hard problems

very quickly. The experimental study we conducted on the PSP and TSPTW showed the interest of that approach. In the PSP case, it showed that a simple DD-LNS solver was competitive with a much more complex, state-of-the-art MIP model that had been tuned by experts in over a decade. The results on TSPTW were just as convincing since our simple DD-LNS solver was able to find new solutions improving the best known solutions of benchmark instances that have been publicly available for many years.

Finally Chapter 7 discussed some prospective material. In particular, it presented a global minded scheme for the compilation of restricted DD when solving hard problem instances. It also showed that this technique might have the potential to improve the quality of the solutions derived from restricted DDs.

8.1 Perspectives for Further Research

At the end of this thesis, it is good to take a step back and reflect on the accomplished work and the perspectives it opens. The coming paragraphs discuss opportunities for future research in the continuation of this thesis, some of which have already been outlined in the previous chapters.

8.1.1 Generalization of the Global Minded Restriction DD

As it has already been mentioned at the beginning of Chapter 7, it would be useful to investigate the effectiveness of the proposed technique on more problems than the PSP. For instance, the TSP is also an example that naturally comes to mind and was used as an illustrative example in the introdution to that same chapter. Variants of those problems such as the TSPTW, or the time dependent TSP (TD-TSP) would seem more interesting through. Indeed, these problems are notorious for being both very hard to solve and relevant to solve real life problems [RCS20; ÇUA21; Vu+20]. Moreover, the formulation of these problems naturally lends itself to different state-space reductions. For instance, one could try to cluster the cities on a geographic basis, on a time window and time-dependent cost increment... or a combination of all of the above.

Another aspect which might be worth investigating would be the impact of various compression/decompression schemes on the effectiveness of the overall method. For instance, when a solution was "inflated" in Chapter 7, it only considered the possible next items that corresponded to the members of the successor cluster in the compressed optimal solution. Another approach that could have been attempted reduces the state space by aggregating the time slots rather than the items, and then inflating the optimal compressed solution by scheduling contiguous batches of the same items.

8.1.2 Investigate Alternate Merge Schemes

In Chapter 2, algorithm 4 presented a procedure to relax an overly large layer. To do so, that procedure heuristically selects the least promising nodes of the layer and merges them into *one* fresh node $\mathcal{M}$ standing for all of them. With that setup, one can hypothesize that:

- a lot of information is lost in the merger and
- the bounds derived from a relaxed DD compiled with this procedure *might* be tighter if the relaxation were to merge fewer nodes at once; thereby creating several relaxed nodes $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ instead of only one.

Figure **??** illustrates how it could be done. It uses the same example of DD that was used in figure 2.2 from Chapter 2. However, rather than depicting an exact, restricted and relaxed version of the same DD, it shows an exact (a) and two relaxed versions of the same DD. Among these two relaxed versions, (b) is compiled as is usually the case with top-down compilation, by merging all the supernumerary nodes of the third layer into one single relaxed node $\mathcal{M}$. The rightmost version (c) uses an alternate scheme to enforce the maximum width of the DD. Instead of combining nodes d, e, and h to create $\mathcal{M}$, it merges d with e and g with h.

The potential benefits of that approach all stem from the possible lesser loss of information during the merge process. This could mean that the relaxed DD is closer to its exact counterpart and hence yields a better upper bound. This is exemplified in Figure **??**: the longest paths in both (a) and (c) have a length of 25 whereas the longest path of (b) is 26. In that example, the 25 happens to be both the value of the optimal solution and the upper bound derived from (c). The lesser loss of information caused by the alternate node merging scheme could also prove useful when it has no impact on the global upper bound derived from a relaxed DD. Indeed, even when the longest path of the DD might not have benefitted from it, other parts of the DD might have. It follows that the local bounds attached to the nodes in the exact cutset of the DD might be much tighter and hence lead to their pruning because of the current best solution.

It is interesting to note that the spirit of the method proposed by Bergman et al. in [Ber+16b] does not prevent the adoption of such a merge scheme. It is worth mentioning that the split-based approach which is used for the compilation of relaxed DD by separation does create relaxed layers having multiple nodes standing for undeveloped exact nodes. This, however, has never been tried when compiling DD with a top-down approach. The simplicity of the top-down compilation combined with a fast vector-based implementation of the procedure could make the derivation of these tight bounds fairly fast to

compute; and hence, yield a significant performance boost to solvers based on that technique.

8.1.3 Distributed parallelism

In Chapter 5 we corroborated the observations of [Ber+14c] stressing the importance of parallel computing when it comes to solving hard combinatorial problems with branch-and-bound DD. In Chapter 4 we explained that our library *ddo* uses a coarse-grained model with shared-memory threads which turned out to be highly efficient. Nevertheless, this model has been shown to fail to scale to the point where it really becomes massively parallel. To lift the systems + hardware limitation on the number of nodes that are currently processed, one could seek to create a distributed solver where multiple machines share their loads and concur to solving the problem. This is easier said than done, however. Indeed, the implementers of such a distributed solver would have to answer numerous difficult questions. For instance: When should the new bounds be communicated to the other workers? How is the load sharing operated? Does each worker maintain its own frontier? If yes, is there a way to limit thrashing by avoiding the repeated expansion of multiple copies of one same node? If not, how and when are the nodes from an exact cutset communicated to the others to avoid saturating the other nodes and/or infrastructure?

8.1.4 Integration with Other Kinds of Solvers

Given that DDO, Constraint Programming (CP), and Mixed Integer Programming (MIP) share strong ties and are often used to solve the same problems, it would make sense to investigate opportunities to hybridize DDO with these other types of solvers. For instance, to integrate the strength of DDO and CP, one would have to strike a delicate tradeoff between conflicting approaches. Among these, let us point out that most CP solvers rely on a trailing mechanism that cannot be safely used in the context of parallel computing. To make the most of their efficient trail implementation, these solvers often also rely on the assumption that the search is performed in a depth-first search manner; which conflicts with the breadth-first approach which is used in the top-down compilation of MDD. Still, the potential benefits are huge in terms of the expressiveness of the model (DP with additional constraints) and performance. Among others because the strong filtering power of domain consistent propagators could significantly reduce the size of the compiled DD and hence strengthen the bounds derived from these DD.

Bibliography

[Ahu+02] Ravindra K Ahuja et al. "A survey of very large-scale neighborhood search techniques". In: *Discrete Applied Mathematics* 123.1-3 (2002), pp. 75–102.

[And+07] H. R. Andersen et al. "A Constraint Store Based on Multivalued Decision Diagrams". In: *Principles and Practice of Constraint Programming*. Ed. by Christian Bessière. Vol. 4741. LNCS. Springer, 2007, pp. 118–132.

[Asc96] Norbert Ascheuer. "Hamiltonian path problems in the on-line optimization of flexible manufacturing systems". PhD thesis. Zuse Institute Berlin, 1996.

[AV06] David Arthur and Sergei Vassilvitskii. *k-means++: The advantages of careful seeding*. Tech. rep. Stanford, 2006.

[BBH11] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V Hicks. "Clique relaxations in social network analysis: The maximum k-plex problem". In: *Operations Research* 59.1 (2011), pp. 133–142.

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

[BC16a] David Bergman and Andre A. Cire. "Multiobjective Optimization by Decision Diagrams". In: *Principles and Practice of Constraint Programming*. Ed. by Michel Rueher. Vol. 9892. LNCS. Springer, 2016, pp. 86–95.

[BC16b] David Bergman and Andre A. Cire. "Theoretical insights and algorithmic tools for decision diagram-based optimization". In: *Constraints* 21.4 (2016), 533–556.

[BC17] David Bergman and Andre A. Cire. "On Finding the Optimal BDD Relaxation". In: *Integration of AI and OR Techniques in Constraint Programming*. Ed. by Domenico Salvagnin and Michele Lombardi. Vol. 10335. LNCS. Springer, 2017, pp. 41–50.

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

[Ber+14a] David Bergman et al. "BDD-based heuristics for binary optimization". In: *Journal of Heuristics* 20.2 (2014), pp. 211–234.

[Ber+14b] David Bergman et al. "Optimization Bounds from Binary Decision Diagrams". In: *INFORMS Journal on Computing* 26.2 (2014), pp. 253–268.

[Ber+14c] David Bergman et al. "Parallel combinatorial optimization with decision diagrams". In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems* (2014), pp. 351–367.

[Ber+16a] David Bergman et al. *Decision Diagrams for Optimization.* Ed. by Barry O'Sullivan and Michael Wooldridge. Springer, 2016.

[Ber+16b] David Bergman et al. "Discrete Optimization with Decision Diagrams". In: *INFORMS Journal on Computing* 28.1 (2016), pp. 47–66.

[BHH11] David Bergman, Willem-Jan van Hoeve, and John N Hooker. "Manipulating MDD relaxations for combinatorial optimization". In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems.* Springer. 2011, pp. 20–35.

[BHM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability.* Vol. 185. IOS press, 2009.

[Bjö+20] Gustav Björdal et al. "Solving satisfaction problems using large-neighbourhood search". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2020, pp. 55–71.

[BMR12] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. "New state-space relaxations for solving the traveling salesman problem with time windows". In: *INFORMS Journal on Computing* 24.3 (2012), pp. 356–371.

[Bur+92] J.R. Burch et al. "Symbolic model checking: 10^{20} States and beyond". In: *Information and Computation* 98.2 (1992), pp. 142–170.

[CD87] KC Chang and DH-C Du. "Efficient algorithms for layer assignment problem". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.1 (1987), pp. 67–78.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[CGS22] Vianney Coppé, Xavier Gillard, and Pierre 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.

[Cir14] Andre A Cire. "Decision Diagrams for Optimization". PhD thesis. Carnegie Mellon University Tepper School of Business, 2014.

[CKC83] Ruen-Wu Chen, Yoji Kajitani, and Shu-Park Chan. "A graph-theoretic via minimization algorithm for two-layer printed circuit boards". In: *IEEE transactions on circuits and systems* 30.5 (1983), pp. 284–299.

[CKV13] M. Emre Celebi, Hassan A. Kingravi, and Patricio A. Vela. "A comparative study of efficient initialization methods for the k-means clustering algorithm". In: *Expert Systems with Applications* 40.1 (2013), pp. 200–210.

[CMT81] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. "State-space relaxation procedures for the computation of bounds to routing problems". In: *Networks* 11.2 (1981), pp. 145–164.

[Cor+09] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[ÇUA21] Esra Çakir, Ziya Ulukan, and Tankut Acarman. "Shortest Fuzzy Hamiltonian Cycle on Transportation Network Using Minimum Vertex Degree and Time-dependent Dijkstra's Algorithm". In: *IFAC-PapersOnLine* 54.2 (2021), pp. 348–353.

[Cus08] James Cussens. "Bayesian network learning by compiling to weighted MAX-SAT". In: *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence (UAI 2008)*. AUAI Press. 2008, pp. 105–112.

[CVH13] Andre A Cire and Willem-Jan Van Hoeve. "Multivalued decision diagrams for sequencing problems". In: *Operations Research* 61.6 (2013), pp. 1411–1428.

[DCS18] Emir Demirović, Geoffrey Chu, and Peter J Stuckey. "Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2018, pp. 99–108.

[Der16] Laurent Deroussi. *Métaheuristiques pour la logistique*. Vol. 2. ISTE Group, 2016.

[DH18] Danial Davarnia and Willem-Jan van Hoeve. "Outer approximation for integer nonlinear programs via decision diagrams". In: *Mathematical Programming* 187.1 (2018), pp. 111–150.

[DSU10] Rodrigo Ferreira Da Silva and Sebastián Urrutia. "A general VNS heuristic for the traveling salesman problem with time windows". In: *Discrete Optimization* 7.4 (2010), pp. 203–211.

[Dum+95] Yvan Dumas et al. "An optimal algorithm for the traveling salesman problem with time windows". In: *Operations research* 43.2 (1995), pp. 367–371.

[Ebl+12] John D Eblen et al. "The maximum clique enumeration problem: algorithms, applications, and implementations". In: *BMC bioinformatics*. Vol. 13. 10. Springer. 2012, pp. 1–11.

[ER59] P. Erdös and A. Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[ESB99] Jubin Edachery, Arunabha Sen, and Franz J Brandenburg. "Graph clustering using distance-k cliques". In: *International Symposium on Graph Drawing*. Springer. 1999, pp. 98–106.

[Fes+02] Paola Festa et al. "Randomized heuristics for the MAX-CUT problem". In: *Optimization methods and software* 17.6 (2002), pp. 1033–1058.

[FL10] Matteo Fischetti and Andrea Lodi. "Heuristics in mixed integer programming". In: *Wiley Encyclopedia of Operations Research and Management Science* (2010).

[Fou22] The Rust Foundation. *The Rust Programming Language*. `https://www.rust-lang.org/`. 2022-04-13.

[Gen+98] Michel Gendreau et al. "A generalized insertion heuristic for the traveling salesman problem with time windows". In: *Operations Research* 46.3 (1998), pp. 330–335.

[GJS74] Michael R Garey, David S Johnson, and Larry Stockmeyer. "Some simplified NP-complete problems". In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. 1974, pp. 47–63.

[GMH20] Rebecca Gentzel, Laurent Michel, and Willem-Jan van Hoeve. "HADDOCK: A language and architecture for decision diagram compilation". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2020, pp. 531–547.

[Gon+20] Jaime E Gonzalez et al. “Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem”. In: *Constraints* (2020), pp. 1–24.

[GSC20a] X. Gillard, P. Schaus, and V. Coppé. *Ddo, a generic and efficient framework for MDD-based optimization.* International Joint Conference on Artificial Intelligence (IJCAI-20); DEMO track. 2020.

[GW94] Ian P Gent and Toby Walsh. “The SAT phase transition”. In: *ECAI.* Vol. 94. PITMAN. 1994, pp. 105–109.

[GW99] Ian P Gent and Toby Walsh. “CSPLib: a benchmark library for constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1999, pp. 480–481.

[HHH10] Samid Hoda, Willem-Jan van Hoeve, and John 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.

[HHT08] T Hadžić, JN Hooker, and P Tiedemann. “Propagating separable equalities in an MDD store”. In: *CPAIOR.* 2008, pp. 318–322.

[HM09] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search.* The MIT press, 2009.

[Hoo06] JN Hooker. “Discrete global optimization with binary decision diagrams”. In: *GICOLAG 2006* (2006).

[Hoo13] J. N. Hooker. “Decision Diagrams and Dynamic Programming”. In: *Integration of AI and OR Techniques in Constraint Programming*. Ed. by Carla Gomes and Meinholf Sellmann. Vol. 7874. LNCS. Springer, 2013, pp. 94–110.

[Hoo17] J. N. Hooker. “Job Sequencing Bounds from Decision Diagrams”. In: *Principles and Practice of Constraint Programming*. Ed. by J. Christopher Beck. Vol. 10416. LNCS. 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 Thomas Schiex and Simon de Givry. Vol. 11802. LNCS. Springer, 2019, pp. 268–283.

[Hor+21] Matthias Horn et al. “A*-based construction of decision diagrams for a prize-collecting scheduling problem”. In: *Computers & Operations Research* 126 (2021), p. 105125.

[HS04] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications.* Elsevier, 2004.

[HVHH10] Samid Hoda, Willem-Jan Van Hoeve, and John 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.

[Ign+14] Alexey Ignatiev et al. "Progression in Maximum Satisfiability." In: *ECAI*. 2014, pp. 453–458.

[ISO21] Nicolas ISOART. "Le problême du voyageur de commerce en programmation par contraintes". Université Côte d'Azur, 2021.

[JVH11] Siddhartha Jain and Pascal Van Hentenryck. "Large neighborhood search for dial-a-ride problems". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2011, pp. 400–413.

[Lab+18] Philippe Laborie et al. "IBM ILOG CP optimizer for scheduling". In: *Constraints* 23.2 (2018), pp. 210–250.

[Lan+93] André Langevin et al. "A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows". In: *Networks* 23.7 (1993), pp. 631–640.

[LIB10] Manuel López-Ibáñez and Christian Blum. "Beam-ACO for the travelling salesman problem with time windows". In: *Computers & operations research* 37.9 (2010), pp. 1570–1583.

[LIB20] Manuel López-Ibáñez and Christian Blum. *Benchmark Instances for the Travelling Salesman Problem with Time Windows*. Online. 2020.

[LK73] Shen Lin and Brian W Kernighan. "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations research* 21.2 (1973), pp. 498–516.

[MD96] Chryssi Malandraki and Robert B Dial. "A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem". In: *European Journal of Operational Research* 90.1 (1996), pp. 45–55.

[MSVH21] L. Michel, P. Schaus, and P. Van Hentenryck. "MiniCP: a lightweight solver for constraint programming". In: *Mathematical Programming Computation* 13.1 (2021), pp. 133–184.

[Net+07] Nicholas Nethercote et al. "MiniZinc: Towards a standard CP modelling language". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[OH19] Ryan J. O'Neil and Karla 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.

[OT07] Jeffrey W Ohlmann and Barrett W Thomas. "A compressed-annealing heuristic for the traveling salesman problem with time windows". In: *INFORMS Journal on Computing* 19.1 (2007), pp. 80–90.

[Ove22] Stack Overflow. *Stack Overflow Developer Survey*. `https://insights.stackoverflow.com/survey/2021`. 2022-04-13.

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

[PD07] Knot Pipatsrisawat and Adnan Darwiche. "A lightweight component caching scheme for satisfiability solvers". In: *International conference on theory and applications of satisfiability testing*. Springer. 2007, pp. 294–299.

[Pes+98] Gilles Pesant et al. "An exact constraint logic programming algorithm for the traveling salesman problem with time windows". In: *Transportation Science* 32.1 (1998), pp. 12–29.

[Pin84] Ron Y Pinter. "Optimal layer assignment for interconnect". In: *Advances in VLSI and Computer Systems* 1.2 (1984), pp. 123–137.

[PR15] Guillaume Perez and Jean-Charles Régin. "Efficient operations on MDDs for building constraint programming models". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15).* 2015, pp. 374–380.

[PW06] Yves Pochet and Laurence A. Wolsey. *Production Planning by Mixed Integer Programming*. Springer, 2006.

[RCR18] Michael Römer, Andre A Cire, and Louis-Martin 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.

[RCS20] Omar Rifki, Nicolas Chiabaut, and Christine Solnon. "On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours". In: *Transportation Research Part E: Logistics and Transportation Review* 142 (2020), p. 102085.

[Rog+91] David F Rogers et al. "Aggregation and disaggregation techniques and methodology in optimization". In: *Operations Research* 39.4 (1991), pp. 553–582.

[Sav85] Martin WP Savelsbergh. "Local search in routing problems with time windows". In: *Annals of Operations research* 4.1 (1985), pp. 285–305.

[Sha98] Paul Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *International conference on principles and practice of constraint programming*. Springer. 1998, pp. 417–431.

[Tab14] John Tabak. *Geometry: the language of space and form*. Infobase Publishing, 2014.

[TH19] Christian Tjandraatmadja and Willem-Jan van Hoeve. "Target Cuts from Relaxed Decision Diagrams". In: *INFORMS Journal on Computing* 31.2 (2019), pp. 285–301.

[Tja18] Christian Tjandraatmadja. "Decision Diagram Relaxations for Integer Programming". PhD thesis. Carnegie Mellon University Tepper School of Business, 2018.

[Tse68] G Tseitin. "On the complexity ofderivation in propositional calculus". In: *Studies in Constrained Mathematics and Mathematical Logic* (1968).

[VLS18] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. "Compact-MDD: Efficiently Filtering (s) MDD Constraints with Reversible Sparse Bit-sets." In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*. 2018, pp. 1383–1389.

[Vu+20] Duc Minh Vu et al. "Dynamic discretization discovery for solving the time-dependent traveling salesman problem with time windows". In: *Transportation Science* 54.3 (2020), pp. 703–720.

[XQ02] Chen Xiong and Wu Qidi. "Formulating the steel scheduling problem as a TSPTW". In: *Proceedings of the 4th World Congress on Intelligent Control and Automation (Cat. No.02EX527)*. Vol. 3. 2002, 1744–1748 vol.3.

[Zip80] Paul H Zipkin. "Bounds for aggregating nodes in network problems". In: *Mathematical programming* 19.1 (1980), pp. 155–177.