Artificial Intelligence: Foundations, Theory, and Algorithms

David Bergman
Andre A. Cire
Willem-Jan van Hoeve
John Hooker

Decision Diagrams for Optimization

Artificial Intelligence: Foundations, Theory, and Algorithms

Series editors

Barry O'Sullivan, Cork, Ireland
Michael Wooldridge, Oxford, UK

More information about this series at http://www.springer.com/series/13900

David Bergman · Andre A. Cire
Willem-Jan van Hoeve · John Hooker

Decision Diagrams for Optimization

Springer

David Bergman
Department of Operations and Information Management, School of Business
University of Connecticut
Storrs, CT
USA

Andre A. Cire
Department of Management, UTSC
University of Toronto
Toronto, ON
Canada

Willem-Jan van Hoeve
Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA
USA

John Hooker
Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA
USA

ISSN 2365-3051 ISSN 2365-306X (electronic)
Artificial Intelligence: Foundations, Theory, and Algorithms
ISBN 978-3-319-42847-5 ISBN 978-3-319-42849-9 (eBook)
DOI 10.1007/978-3-319-42849-9

Library of Congress Control Number: 2016953636

© Springer International Publishing Switzerland 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

1 Introduction 1
1.1 Motivation for the Book 1
1.2 A New Solution Technology 2
1.3 An Example 4
1.4 Plan of the Book 8

2 Historical Overview 11
2.1 Introduction 11
2.2 Origins of Decision Diagrams 12
2.3 Decision Diagrams in Optimization 15
2.3.1 Early Applications 15
2.3.2 A Discrete Optimization Method 16
2.3.3 Decision Diagrams in Constraint Programming 17
2.3.4 Relaxed Decision Diagrams 18
2.3.5 A General-Purpose Solver 19
2.3.6 Markov Decision Processes 21

3 Exact Decision Diagrams 23
3.1 Introduction 23
3.2 Basic Definitions 24
3.3 Basic Concepts of Decision Diagrams 25
3.4 Compiling Exact Decision Diagrams 27
3.4.1 Dynamic Programming 28
3.4.2 Top-Down Compilation 30
3.5 Maximum Independent Set Problem 32

3.6 Set Covering Problem . . . 34
3.7 Set Packing Problem . . . 37
3.8 Single-Machine Makespan Minimization . . . 39
3.9 Maximum Cut Problem . . . 42
3.10 Maximum 2-Satisfiability Problem . . . 44
3.11 Compiling Decision Diagrams by Separation . . . 46
3.12 Correctness of the DP Formulations . . . 50

4 Relaxed Decision Diagrams . . . 55
4.1 Introduction . . . 55
4.2 Top-Down Compilation of Relaxed DDs . . . 57
4.3 Maximum Independent Set . . . 59
4.4 Maximum Cut Problem . . . 60
4.5 Maximum 2-Satisfiability Problem . . . 63
4.6 Computational Study . . . 64
4.6.1 Merging Heuristics . . . 64
4.6.2 Variable Ordering Heuristic . . . 65
4.6.3 Bounds vs. Maximum BDD Width . . . 66
4.6.4 Comparison with LP Relaxation . . . 67
4.7 Compiling Relaxed Diagrams by Separation . . . 74
4.7.1 Single-Machine Makespan Minimization . . . 76

5 Restricted Decision Diagrams . . . 83
5.1 Introduction . . . 83
5.2 Top-Down Compilation of Restricted DDs . . . 85
5.3 Computational Study . . . 86
5.3.1 Problem Generation . . . 87
5.3.2 Solution Quality and Maximum BDD Width . . . 89
5.3.3 Set Covering . . . 90
5.3.4 Set Packing . . . 92

6 Branch-and-Bound Based on Decision Diagrams . . . 95
6.1 Introduction . . . 95
6.2 Sequential Branch-and-Bound . . . 96
6.3 Exact Cutsets . . . 97
6.4 Enumeration of Subproblems . . . 98
6.4.1 Exact Cutset Selection . . . 100

6.5 Computational Study ... 100
6.5.1 Results for the MISP ... 101
6.5.2 Results for the MCP ... 104
6.5.3 Results for MAX-2SAT ... 108
6.6 Parallel Branch-and-Bound ... 109
6.6.1 A Centralized Parallelization Scheme ... 112
6.6.2 The Challenge of Effective Parallelization ... 113
6.6.3 Global and Local Pools ... 113
6.6.4 Load Balancing ... 114
6.6.5 DDX10: Implementing Parallelization Using X10 ... 116
6.6.6 Computational Study ... 116

7 Variable Ordering ... 123
7.1 Introduction ... 123
7.2 Exact BDD Orderings ... 125
7.3 Relaxed BDD Orderings ... 130
7.4 Experimental Results ... 131
7.4.1 Exact BDDs for Trees ... 132
7.4.2 Exact BDD Width Versus Relaxation BDD Bound ... 132
7.4.3 Relaxation Bounds ... 134

8 Recursive Modeling ... 137
8.1 Introduction ... 137
8.2 General Form of a Recursive Model ... 139
8.3 Examples ... 141
8.3.1 Single-Machine Scheduling ... 141
8.3.2 Sequence-Dependent Setup Times ... 142
8.3.3 Minimum Bandwidth ... 144
8.4 State-Dependent Costs ... 146
8.4.1 Canonical Arc Costs ... 147
8.4.2 Example: Inventory Management ... 151
8.5 Nonserial Recursive Modeling ... 153

9 MDD-Based Constraint Programming ... 157
9.1 Introduction ... 157
9.2 Constraint Programming Preliminaries ... 158
9.3 MDD-Based Constraint Programming ... 164

9.3.1 MDD Propagation . . . 166
9.3.2 MDD Consistency . . . 167
9.3.3 MDD Propagation by Intersection . . . 169
9.4 Specialized Propagators . . . 173
9.4.1 Equality and Not-Equal Constraints . . . 173
9.4.2 Linear Inequalities . . . 174
9.4.3 Two-Sided Inequality Constraints . . . 174
9.4.4 ALLDIFFERENT Constraint . . . 175
9.4.5 AMONG Constraint . . . 176
9.4.6 ELEMENT Constraint . . . 177
9.4.7 Using Conventional Domain Propagators . . . 178
9.5 Experimental Results . . . 178

10 MDD Propagation for SEQUENCE Constraints . . . 183
10.1 Introduction . . . 183
10.2 MDD Consistency for SEQUENCE Is NP-Hard . . . 186
10.3 MDD Consistency for SEQUENCE Is Fixed Parameter Tractable . . . 189
10.4 Partial MDD Filtering for SEQUENCE . . . 190
10.4.1 Cumulative Sums Encoding . . . 191
10.4.2 Processing the Constraints . . . 192
10.4.3 Formal Analysis . . . 194
10.5 Computational Results . . . 196
10.5.1 Systems of SEQUENCE Constraints . . . 198
10.5.2 Nurse Rostering Instances . . . 201
10.5.3 Comparing MDD Filtering for SEQUENCE and AMONG . . . 203

11 Sequencing and Single-Machine Scheduling . . . 205
11.1 Introduction . . . 205
11.2 Problem Definition . . . 207
11.3 MDD Representation . . . 208
11.4 Relaxed MDDs . . . 211
11.5 Filtering . . . 214
11.5.1 Filtering Invalid Permutations . . . 214
11.5.2 Filtering Precedence Constraints . . . 215
11.5.3 Filtering Time Window Constraints . . . 215
11.5.4 Filtering Objective Function Bounds . . . 216
11.6 Inferring Precedence Relations from Relaxed MDDs . . . 218

11.7 Refinement 219
11.8 Encoding Size for Structured Precedence Relations 221
11.9 Application to Constraint-Based Scheduling 222
11.9.1 Experimental Setup 223
11.9.2 Impact of the MDD Parameters 224
11.9.3 Traveling Salesman Problem with Time Windows 227
11.9.4 Asymmetric Traveling Salesman Problem with Precedence Constraints 228
11.9.5 Makespan Problems 230
11.9.6 Total Tardiness 233

References 235

Index 249

Foreword

This book provides an excellent demonstration of how the concepts and tools of one research community can cross into another, yielding powerful insights and ideas.

Early work on decision diagrams focused on modeling and verifying properties of digital systems, including digital circuits and abstract protocols. Decision diagrams (DDs) provided a compact representation of these systems and a useful data structure for algorithms to construct these representations and to answer queries about them. Fundamentally, though, they were used to solve problems having yes/no answers, such as: "Is it possible for the system to reach a deadlocked state?", or "Do these two circuits compute the same function?"

Using DDs for optimization introduces an entirely new set of possibilities and challenges. Rather than just finding some satisfying solution, the program must find a "best" solution, based on some objective function. Researchers in the digital systems and verification communities recognized that, given a DD representation of a solution space, it is easy to count the number of solutions and to find an optimal solution based on very general classes of objective functions. But, it took the skill of leading experts in optimization, including the authors of this book, to fully expand DDs into a general-purpose framework for solving optimization problems.

The authors show how the main strategies used in discrete optimization, including problem relaxation, branching search, constraint propagation, primal solving, and problem-specific modeling, can be adapted and cast into a DD framework. DDs become a data structure for managing the entire optimization process: finding upper bounds and feasible solutions, storing solutions to subproblems, applying global constraints, and guiding further search. They are especially effective for solving problems that fare poorly with traditional optimization techniques, including linear

programming relaxation, such as ones having combinatorial constraints and non-convex cost functions.

Over the 10 years in which the authors have been developing these ideas, they have transformed DDs well beyond what has been used by the verification community. For example, whereas most DD-based verification techniques build the representations from the bottom up, based on a symbolic execution of the system description, the authors build their DDs from the top down, based on a direct encoding of the solution space. Some of the ideas presented here have analogs in the verification world, for example the idea of restricted and relaxed DDs are similar to the abstraction-refinement approaches used in verification. Others, however, are strikingly new. Perhaps some of these ideas could be transferred back to the verification community to increase the complexity and classes of systems they can verify.

Pittsburgh, USA, August 2016 *Randal E. Bryant*

Chapter 1
Introduction

Abstract This introductory chapter explains the motivation for developing decision diagrams as a new discrete optimization technology. It shows how decision diagrams implement the five main solution strategies of general-purpose optimization and constraint programming methods: relaxation, branching search, constraint propagation, primal heuristics, and intelligent modeling. It presents a simple example to illustrate how decision diagrams can be used to solve an optimization problem. It concludes with a brief outline of the book.

1.1 Motivation for the Book

Optimization is virtually ubiquitous in modern society. It determines how flight crews are scheduled, how ads are displayed on web sites, how courier services route packages, how banks manage investments, and even the order of songs played on online radio. Spurred by the increasing availability of data and computer resources, the variety of applications promises to grow even more rapidly in the future.

One reason for this trend is steady improvement in optimization methods. We have seen dramatic reductions in solution times, not only for techniques specialized to particular problems, but even more markedly for general-purpose mathematical programming and constraint programming solvers. Problems that took hours or days to solve in the past now solve in seconds, due more to algorithmic advancements than to increases in computer power [33].

Despite these advances, a wide range of problems remain beyond the reach of generic optimization solvers. There are various reasons for this. The problems may

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_1

contain a combinatorial structure that is hard to exploit by existing techniques. They may involve cost functions and constraints that are too difficult to formulate for general-purpose solvers. Most importantly, human activities are becoming rapidly more complex and integrated. They pose optimization problems that often grow too rapidly for solvers to keep up. The integer or constraint programming models for these problems may be too large even to load into an existing solver. These reasons alone suffice to motivate research on new general-purpose optimization technologies that accommodate alternative modeling paradigms.

1.2 A New Solution Technology

We present a solution technology based on *decision diagrams*, which have recently brought a new perspective to the field of discrete optimization. A decision diagram is a graphical data structure originally used to represent Boolean functions [3, 110], with successful applications in circuit design and formal verification [37, 99]. Decision diagrams were only recently introduced into optimization and constraint programming [4, 19, 81, 83, 86, 108, 157], and they are already showing potential as an alternative to existing methods. They provide new approaches to implementing the five primary solution strategies of general-purpose methods: relaxation, branching search, constraint propagation, primal heuristics, and modeling to exploit problem structure. These new approaches can enhance existing solvers or provide the basis for a solver based entirely on decision diagrams.

Relaxation is an essential element of general-purpose optimization methods, particularly for mathematical programming problems. It most often takes the form of a continuous relaxation, perhaps strengthened by cutting planes, and it almost always requires a problem formulation in terms of inequality constraints, preferably using linear (or at least convex) expressions. Relaxation is key because the optimal value of a relaxation is a bound on the optimal value of the original problem, and a good bound is often indispensable to keeping the search time within practical limits. Decision diagrams can provide an equally useful *discrete* relaxation of the problem. While one can, in principle, build a decision diagram that exactly represents the problem, the construction procedure can be modified to yield a much smaller diagram that represents a relaxation of the problem. The length of a shortest (or longest) path in this diagram is an easily computed bound on the optimal value of the original problem. This approach has the advantage that it does not rely on inequality

formulations, linearity, convexity, or even closed-form expressions, but exploits quite different properties of the problem. Decision diagrams may therefore benefit from combinatorial structures that have proved intractable for existing methods.

Branching search is ubiquitous in general-purpose solvers. Decision diagrams enable a type of branching search that is significantly different from conventional schemes and could prove more efficient. Rather than branch on the values of a variable, one branches on nodes of a relaxed decision diagram. This in effect enumerates structured pools of solutions, reduces symmetry, and takes advantage of information about the problem that is encoded in the relaxed decision diagram. The bounds provided by relaxed diagrams limit the complexity of the search, resulting in a branch-and-bound algorithm somewhat analogous to those used in mixed-integer programming but relying on bounds obtained from shortest paths in the branching structure itself. In addition, computational testing to date indicates that branching in a decision diagram parallelizes much more effectively than search algorithms in mixed-integer programming solvers.

Constraint propagation is an essential tool for constraint programming solvers. Specialized algorithms exploit the structure of high-level constraints in the problem (known as "global constraints") to exclude values that individual variables cannot assume in any feasible solution. These reduced variable domains are passed on (propagated) to other constraints, where they can be reduced further. Decision diagrams enable a potentially more powerful form of constraint propagation. Rather than transmit a limited amount of information through individual variable domains, one can transmit much more information through a relaxed decision diagram that carries relationships between variables. This, in turn, can reduce the search substantially by identifying infeasible subtrees before they are explored.

Primal heuristics are methods for finding feasible solutions of the problem. They are contrasted with "dual" methods (such as solving a relaxation), which find bounds for proving optimality. A variety of primal heuristics have proved essential to the speedups that general-purpose solvers have exhibited in the past few years. Decision diagrams again provide an attractive alternative, because *restricted* diagrams (the opposite of relaxed diagrams) provide a very competitive primal heuristic that is fast, based on a single technology, and simple to implement. A restricted diagram is constructed by judiciously omitting nodes from an exact diagram as it is constructed, and any shortest path in the resulting diagram corresponds to a good feasible solution.

Intelligent *modeling* is often a prerequisite to obtaining a solution in reasonable time. In mixed-integer programming, one uses an inequality formulation that ideally has a tight linear relaxation and perhaps contains redundant valid inequalities that further tighten the relaxation. In constraint programming, one selects variables and global constraints that best capture the structure of the problem and lead to effective propagation. Decision diagrams, by contrast, are based on recursive modeling very similar to that used in deterministic dynamic programming. One must define state variables, but any constraint or cost function definable in terms of the current state is permissible. This allows one to formulate a wide range of problems that have no practical inequality or constraint-based formulation, and in which linearity or convexity is no longer an issue.

Recursive formulations are generally seen as too problem-specific for implementation in general-purpose solvers. Worse, hand-coded methods must typically enumerate a state space that grows exponentially (the "curse of dimensionality") or make do with a more tractable approximation of it. Decision diagrams offer a possible solution to these dilemmas. They allow modeling flexibility very similar to that of dynamic programming while solving the model with a branch-and-bound method rather than by enumerating the state space, thus possibly ameliorating the "curse" and opening the door to a general-purpose solver for recursive models. An added advantage is that recursive models tend to be compact, thus obviating the necessity of loading a huge constraint-based model into the solver.

1.3 An Example

A small example will illustrate some of the key concepts of optimization with decision diagrams. Consider the integer programming problem

$$
\begin{aligned}
\max \quad & 5x_1 + 3x_2 - x_3 - 15x_4 - 3x_5 \\
\text{subject to} \quad & x_1 + x_2 \geq 1 \\
& x_1 + x_3 + x_5 \leq 2 \\
& x_1 - x_3 - x_5 \leq 0 \\
& -x_1 + x_3 - x_5 \leq 0 \\
& x_1 + 3x_2 - 4x_4 \leq 0 \\
& x_1, \ldots, x_5 \in \{0, 1\}
\end{aligned} \tag{1.1}
$$

The example is chosen for its simplicity, and not because decision diagram technology is primarily directed at integer programming problems. This is only one of many classes of problems that can be formulated recursively for solution by decision diagrams.

A decision diagram for this problem instance represents possible assignments to the variables $x_1, \ldots, x_5$ and is depicted in Fig. 1.1. It is a directed acyclic graph in which the nodes are partitioned into six layers so that an arc leaving a node at layer i corresponds to a value assignment for variable x_i. Since all variables are binary in this problem, the diagram is a *binary decision diagram* and has two types of arcs: dashed arcs in layer i represent the assignment $x_i = 0$, and solid arcs represent $x_i = 1$. Any path from the root node r to the terminal node t represents a complete value assignment to the variables x_i. One can verify that the diagram in Fig. 1.1 exactly represents the seven feasible solutions of problem (1.1).

To capture the objective function, we associate with each arc a *weight* that represents the contribution of that value assignment to the objective function. Dashed arcs have a weight of zero in this instance, while solid arcs have weight equal to the objective function coefficient of that variable. It follows that the value assignment that maximizes the objective function corresponds to the longest path from r to t with respect to these arc weights. For the diagram in Fig. 1.1, the longest path has value -8 and indicates the assignment $(x_1, \ldots, x_5) = (1, 1, 1, 1, 0)$, which is the optimal solution of (1.1). In general, any linear function (or, more generally, any separable function) can be optimized in polynomial time in the size of the diagram. This fact, alongside the potential to represent feasible solutions in a compact fashion, were early motivations for the application of decision diagrams to optimization.

A formidable obstacle to this approach, however, is that a decision diagram that exactly represents the feasible solutions of a problem can grow exponentially

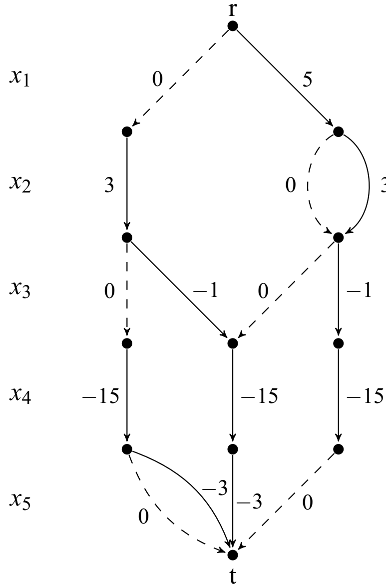


Fig. 1.1 A decision diagram for problem (1.1). Arcs are partitioned into layers, one for each problem variable. Dashed and solid arcs in layer i represent the assignments $x_i = 0$ and $x_i = 1$, respectively.

with the problem size. This is true in particular of integer programming, because a shortest- or longest-path computation in the associated diagram is a linear programming problem, and there are integer programming problems that cannot be reformulated as linear programming problems of polynomial size [65]. In fact, most practical problem classes result in decision diagrams that grow exponentially and thus can be solved only in very small instances.

To circumvent this issue, the authors in [4] introduced the concept of a *relaxed decision diagram*, which is a diagram of limited size that represents an overapproximation of the solution set. That is, all feasible solutions are associated with some path in the diagram, but not all paths in the diagram correspond to a feasible solution of the problem. The size of the diagram is controlled by limiting its *width*, which is the maximum number of nodes in any layer. A key property of relaxed diagrams is that a longest path now yields an upper bound on the maximum value of an objective function (and a shortest path yields a lower bound for minimization problems).

For example, Fig. 1.2 depicts a relaxed decision diagram for problem (1.1) with a limited width of 2. Of the ten *r*–*t* paths, seven represent the feasible solutions of (1.1), and the remaining three represent infeasible solutions. In particular, the

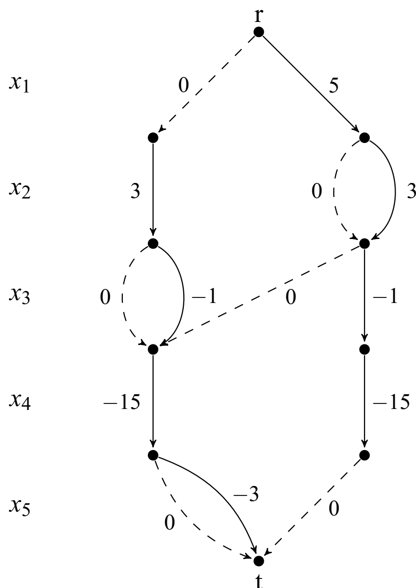


Fig. 1.2 A relaxed decision diagram for problem (1.1), with a limited width of at most two nodes in any layer.

longest path represents an infeasible assignment $(x_1,\ldots,x_5) = (1,1,0,1,0)$ with value -7, which is an upper bound of the optimal solution value, -8. Such a dual bound can be tighter than those provided by other generic technologies. For example, the linear programming relaxation for problem (1.1), obtained by replacing the integrality constraints by $0 \le x_j \le 1$, yields a relatively weak upper bound of 0.25.

Relaxed decision diagrams were initially proposed in [4] as an alternative to the domain store commonly used in constraint programming solvers. It was shown that relaxed diagrams can reveal inconsistent variable assignments that conventional domain propagation fails to detect. For instance, we can deduce from the relaxed decision diagram in Fig. 1.2 that $x_4 = 1$ in any feasible solution, which does not follow from domain consistency maintenance. Generic methods for systematically compiling relaxed diagrams for constraint programming models were developed in [84, 94]. Decision diagrams were first used to obtain optimization bounds in [28], where lower bounds for set covering instances were compared with those obtained by integer programming. In this book, we further develop these techniques and apply them to a wide variety of optimization and constraint programming problems.

1.4 Plan of the Book

After a brief literature review in Chapter 2, the book develops methods for constructing exact, relaxed, and restricted decision diagrams for optimization problems. It then presents a general-purpose method for solving discrete optimization problems, followed by discussions of variable order, recursive modeling, constraint programming, and two special classes of problems.

Chapter 3 formally develops methods for constructing decision diagrams for discrete optimization, based on a recursive (dynamic programming) model of the problem that associates states with nodes of the diagram. It presents recursive models of three classical optimization problems that reappear in later chapters: the maximum independent set problem, the maximum cut problem on a graph, and the maximum 2-satisfiability problem.

Chapter 4 modifies the compilation procedure of the previous chapter to create a relaxed decision diagram by merging states as the diagram is constructed. It investigates how various parameters affect the quality of the resulting bound. It reports computational tests showing that, for the independent set problem, relaxed decision diagrams can deliver tighter bounds, in less computation time, than those obtained by linear programming and cutting planes at the root node in a state-of-the-art integer programming solver.

Chapter 5 presents an algorithm for top-down construction of restricted decision diagrams that provide a primal heuristic for finding feasible solutions. Computational results show that restricted diagrams can deliver better solutions than integer programming technology for large set covering and set packing problems.

Chapter 6 combines the ideas developed in previous chapters to devise a general-purpose solution method for discrete optimization, based entirely on decision diagrams. It introduces a novel search algorithm that branches on nodes of a relaxed or restricted decision diagram. It reports computational tests showing that a solver based on decision diagrams is competitive with or superior to state-of-the-art integer programming technology on the three classes of problems described earlier, even though integer programming benefits from decades of development, and even though these problems have natural integer programming models. Further computational tests indicate that branching in a decision diagram can utilize massively parallel computation much more effectively than integer programming methods.

Chapter 7 examines more deeply the effect of variable ordering on the size of exact decision diagrams and the quality of bounds provided by relaxed diagrams,

with particular emphasis on the maximum independent set problem. It shows, for example, that the width of an exact diagram for this problem is bounded by Fibonacci numbers for an appropriate ordering.

Chapter 8 focuses on the type of recursive modeling that is required for solution by decision diagrams. It presents a formal development that highlights how solution by decision diagrams differs from traditional enumeration of the state space. It then illustrates the versatility of recursive modeling with examples: single-facility scheduling, scheduling with sequence-dependent setup times (as in the traveling salesman problem with time windows), and minimum bandwidth problems. It shows how to represent state-dependent costs with canonical arc costs in a decision diagram, a technique that can sometimes greatly simplify the recursion, as illustrated by a textbook inventory management problem. It concludes with an extension to nonserial recursive modeling and nonserial decision diagrams.

Chapter 9 describes how decision diagrams can enhance constraint programming theory and technology. It presents basic concepts of constraint programming, defines a concept of consistency for decision diagrams, and surveys various algorithms for propagation through decision diagrams. It then presents specialized propagators for particular constraints, including equalities and disequalities, linear inequalities, two-sided inequalities, all-different constraints, among constraints, and element constraints. It concludes with computational results that show the superiority of propagation through decision diagrams relative to traditional propagation through variable domains.

The topic of decision-diagram-based constraint programming is continued in Chapter 10, which considers the "sequence" global constraint in detail. The sequence constraint finds application in, e.g., car manufacturing and nurse rostering problems. The chapter shows that establishing full consistency for this constraint is NP-hard, but also describes a specialized propagator that achieves much stronger results than existing methods that are based on traditional domain propagation.

Chapter 11 focuses on the application of decision diagrams to sequencing and single-machine scheduling problems. In these problems, the goal is to find the best order for performing a set of tasks. Typical examples can be found in manufacturing and routing applications, such as assembly line sequencing and package delivery. The chapter describes how decision diagrams can be used in generic constraint-based scheduling systems. The computational results demonstrate that the added power of decision diagrams can improve the performance of such systems by orders of magnitude.

Chapter 2
Historical Overview

Abstract This chapter provides a brief review of the literature on decision diagrams, primarily as it relates to their use in optimization and constraint programming. It begins with an early history of decision diagrams and their relation to switching circuits. It then surveys some of the key articles that brought decision diagrams into optimization and constraint solving. In particular it describes the development of relaxed and restricted decision diagrams, the use of relaxed decision diagrams for enhanced constraint propagation and optimization bounding, and the elements of a general-purpose solver. It concludes with a brief description of the role of decision diagrams in solving some Markov decision problems in artificial intelligence.

2.1 Introduction

Research on decision diagrams spans more than five decades, resulting in a large literature and a wide range of applications. This chapter provides a brief review of this literature, primarily as it relates to the use of decision diagrams in optimization and constraint programming. It begins with an early history of decision diagrams, showing how they orginated from representations of switching circuits and evolved to the ordered decision diagrams now widely used for circuit design, product configuration, and other purposes.

The chapter then surveys some of the key articles that brought decision diagrams into optimization and constraint programming. It relates how decision diagrams initially played an auxiliary role in the solution of some optimization problems and were subsequently proposed as a stand-alone optimization method, as well as

© Springer International Publishing Switzerland 2016
D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_2

a filtering technique in constraint programming. At this point the key concept of a relaxed decision diagram was introduced and applied as an enhanced progagation mechanism in constraint programming and a bounding technique in optimization. These developments led to a general-purpose optimization algorithm based entirely on decision diagram technology. The chapter concludes with a brief description of the auxiliary role of decision diagrams in solving some Markov decision problems in artificial intelligence.

This discussion is intended as a brief historical overview rather than an exhaustive survey of work in the field. Additional literature is cited throughout the book as it becomes relevant.

2.2 Origins of Decision Diagrams

The basic idea behind decision diagrams was introduced by Lee [110] in the form of a *binary-decision program*, which is a particular type of computer program that represents a switching circuit. Shannon had shown in his famous master's thesis [144] that switching circuits can be represented in Boolean algebra, thus bringing Boole's ideas into the computer age. Lee's objective was to devise an alternative representation that is more conducive to the actual computation of the outputs of switching circuits.

Figure 2.1, taken from Lee's article, presents a simple switching circuit. The switches are controlled by binary variables x, y and z. The symbol x' in the circuit indicates a switch that is open when $x = 0$, while x indicates a switch that is open when $x = 1$, and similarly for the other variables. The output of the circuit is 1 if there is an open path from left to right, and otherwise the output is 0. For instance, $(x,y,z) = (1,1,0)$ leads to an output of 1, while $(x,y) = (0,0)$ leads to an output of 0, irrespective of the value of z.

A binary-decision program consists of a single type of instruction that Lee calls T, which has the form

$$T:\ x;\ A,B.$$

The instruction states: if $x = 0$, go to the instruction at address A, whereas if $x = 1$, go to the instruction at address B. The switching circuit of Fig. 2.1 is represented by the binary-decision program

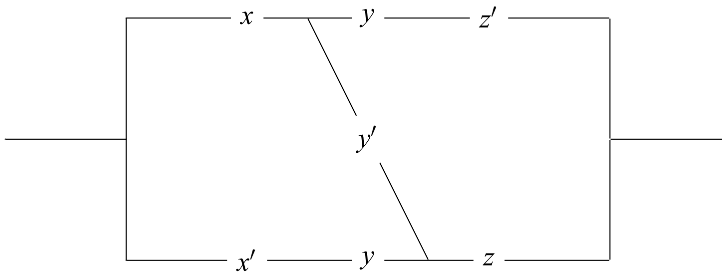


Fig. 2.1 Example of a switching circuit from [110].

$$\begin{array}{l} 1.\;\; T:\; x;\;\; 2,4 \\ 2.\;\; T:\; y;\;\; \theta,3 \\ 3.\;\; T:\; z;\;\; \theta,I \\ 4.\;\; T:\; y;\;\; 3,5 \\ 5.\;\; T:\; z;\;\; I,\theta \end{array} \tag{2.1}$$

where θ is Lee's symbol for an output of 0, and I for an output of 1. The five instructions correspond conceptually to the nodes of a decision diagram, because at each node there is a choice to move to one or two other nodes, and the choice depends on the value of an associated variable. However, the nodes need not be organized into layers that correspond to the variables, and a given assignment to the variables need not correspond to a path in the diagram. A BDD representation of (2.1) appears in Fig. 2.2. In this case, the nodes can be arranged in layers, but there is no path corresponding to $(x,y,z) = (0,0,1)$.[1]

Lee formulated rules for constructing a switching circuit from a binary-decision program. He also provided bounds on the minimum number of instructions that are necessary to represent a given Boolean function. In particular, he showed that computing the output of a switching circuit with a binary-decision program is in general faster than computing it through Boolean operations *and*, *or*, and *sum*, often by orders of magnitude.

The graphical structure we call a binary decision diagram, as well as the term, were introduced by Akers [3]. Binary-decision programs and BDDs are equivalent in some sense, but there are advantages to working with a graphical representation. It is easier to manipulate and provides an implementation-free description of a Boolean function, in the sense that it can be used as the basis for different algorithms

[1] There is such a path, in this case, if one treats the arc from y to 0 as a "long arc," meaning that z can take either value on this arc.

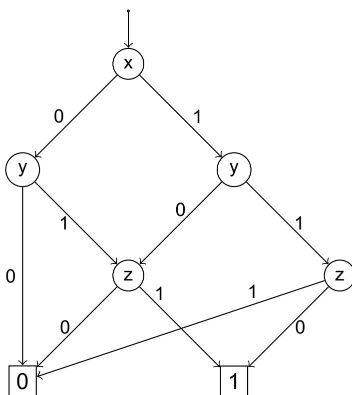


Fig. 2.2 Binary decision diagram corresponding to the binary-decision program (2.1).

for computing outputs. Akers used BDDs to analyze certain types of Boolean functions and as a tool for test generation; that is, for finding a set of inputs which can be used to confirm that a given implementation performs correctly. He also showed that a BDD can often be simplified by superimposing isomorphic portions of the BDD.

The advance that led to the widespread application of BBDs was due to Bryant [37]. He adopted a data structure in which the decision variables are restricted to a particular ordering, forcing all nodes in a layer of the BDD to correspond to the same decision variable. The result is an *ordered* decision diagram (which we refer to simply as a decision diagram in this book). For any given ordering of the variables, all Boolean functions can be represented by ordered BDDs, and many ordered BDDs can be simplified by superimposing isomorphic portions of the BDD. A BDD that can be simplified no further in this fashion is known as a *reduced ordered* binary decision diagram (RO-BDD). A fundamental result is that RO-BDDs provide a canonical representation of Boolean functions. That is, for any given variable ordering, every Boolean function has a unique representation as an RO-BDD. This allows one to check whether a logic circuit implements a desired Boolean function, for example, by constructing an RO-BDD for either and noting whether they are identical.

Another advantage of ordered BDDs is that operations on Boolean functions, such as disjunction and conjunction, can be performed efficiently by an appropriate operation on the corresponding diagrams. The time complexity for an operation is bounded by the product of the sizes of the BDDs. Unfortunately, the BDDs for some popular circuits can grow exponentially even when they are reduced. For example,

the RO-BDD grows linearly for an adder circuit but exponentially for a multiplier circuit. Furthermore, the size of a reduced BDD can depend dramatically on the variable ordering. Computing the ordering that yields the smallest BDD is a co-NP-complete problem [71]. Ordering heuristics that take into account the problem domain may therefore be crucial in obtaining small BDDs for practical applications.

The canonical representation and efficient operations introduced by Bryant led to a stream of BDD-related research in computer science. Several variants of the basic BDD data structure were proposed for different theoretical and practical purposes. A monograph by Wegener [157] provides a comprehensive survey of different BDD types and their uses in practice. Applications of BDDs include formal verification [99], model checking [50], product configuration [5]—and, as we will see, optimization.

2.3 Decision Diagrams in Optimization

Decision diagrams initially played an auxiliary role in optimization, constraint programming, and Markov decision processes. In recent years they have been proposed as an optimization technique in their own right. We provide a brief survey of this work, focusing primarily on early contributions.

2.3.1 Early Applications

One of the early applications of decision diagrams was to *solution counting* in combinatorial problems, specifically to the counting of knight's tours [111]. A BDD is created to represent the set of feasible solutions, as described in the previous chapter. Since the BDD is a directed acyclic graph, the number of solutions can then be counted in linear time (in the size of the BDD) using a simple recursive algorithm. This approach is impractical when the BDD grows exponentially with instance size, as it often does, but in such case BDDs can be combined with backtracking and divide-and-conquer strategies.

Lai, Pedram and Vrudhula [108] used BDDs to represent the feasible sets of 0/1 programming subproblems while a search tree is under construction. Their solution algorithm begins by building a search tree by a traditional branch-and-cut procedure. After some branching rounds, it generates BDDs to represent the

feasible sets of the relatively small subproblems at leaf nodes. The optimal solutions of the subproblems are then extracted from the BDDs, so that no more branching is necessary. Computational experiments were limited to a small number of instances but showed a significant improvement over the IP methods of the time. We remark in passing that Wegener's monograph [157], mentioned earlier, proposes alternative methods for formulating 0/1 programming problems with BDDs, although they have not been tested experimentally. It also studies the growth of BDD representations for various types of Boolean functions.

Hachtel and Somenzi [81] showed how BDDs can help solve maximum flow problems in large-scale 0/1 networks, specifically by enumerating augmenting paths. Starting with a flow of 0, a corresponding flow-augmenting BDD is compiled and analyzed to compute the next flow. The process is repeated until there are no more augmenting paths, as indicated by an empty BDD. Hachtel and Somenzi were able to compute maximum flows for graphs having more than 10^{27} vertices and 10^{36} edges. However, this was only possible for graphs with short augmenting paths, because otherwise the resulting BDDs would be too large.

Behle [19] showed how BDDs can help generate valid inequalities (cutting planes) for general 0/1 programming. He first studied the reduced BDD that encodes the threshold function represented by a 0/1 linear inequality, which he called a *threshold BDD*. He also showed how to compute a variable ordering that minimizes the size of the BDD. To obtain a BDD for a 0/1 programming problem, he conjoined the BDDs representing the individual inequalities in the problem, using an algorithm based on parallel computation. The resulting BDD can, of course, grow quite large and is practical only for small problem instances. He observed that when the BDD is regarded as a flow network, the polytope representing its feasible set is the convex hull of the feasible set of the original 0/1 problem. Based on this, he showed how to generate valid inequalities for the 0/1 problem by analyzing the polar of the flow polytope, a method that can be effective for small but hard problem instances.

2.3.2 A Discrete Optimization Method

Decision diagrams were proposed as a stand-alone method for discrete optimization by Hadžić and Hooker [82, 86], using essentially the approach described in the previous chapter, but initially without the concept of a relaxed diagram. They noted that decision diagrams can grow exponentially but provide two benefits that are

not enjoyed by other optimization methods: (a) they are insensitive to whether the constraint and objective function are linear or convex, which makes them appropriate for global optimization, and (b) they are well suited to comprehensive *postoptimality analysis*.

Postoptimality analysis is arguably important because simply finding an optimal solution misses much of the information and insight encoded in an optimization model. Decision diagrams provide a transparent data structure from which one can quickly extract answers to a wide range of queries, such as how the optimal solution would change if certain variables were fixed to certain values, or what alternative solutions are available if one tolerates a small increase in cost. The power of this analysis is illustrated in [82, 86] for capital budgeting, network reliability, and portfolio design problems.

In subsequent work [83], Hadžić and Hooker proposed a cost-bounding method for reducing the size of the decision diagram used for postoptimality analysis. Assuming that the optimal value is given, they built a BDD that represents all solutions whose cost is within a given tolerance of the optimum. Since nearly all postoptimality analysis of interest is concerned with solutions near the optimum, such a cost-bounded BDD is adequate. They also showed how to reduce the size of the BDD significantly by creating a *sound* cost-bounded BDD rather than an exact one. This is a BDD that introduces some infeasible solutions, but only when their cost is outside the tolerance. When conducting sensitivity analysis, the spurious solutions can be quickly discarded by checking their cost. Curiously, a sound BDD can be substantially smaller than an exact one even though it represents more solutions, provided it is properly constructed. This is accomplished by pruning and contraction methods that remove certain nodes and arcs from the BDD. A number of experiments illustrated the space-saving advantages of sound BDDs.

Due to the tendency of BDDs to grow exponentially, a truly scalable solution algorithm for discrete optimization became available only with the introduction of relaxed decision diagrams. These are discussed in Section 2.3.4 below.

2.3.3 Decision Diagrams in Constraint Programming

Decision diagrams initially appeared in constraint programming as a technique for processing certain *global constraints*, which are high-level constraints frequently used in constraint programming models. An example of a global constraint is

ALLDIFFERENT(X), which requires that the set X of variables take distinct values. Each global constraint represents a specific combinatorial structure that can be exploited in the solution process. In particular, an associated *filtering* algorithm removes infeasible values from variable domains. The reduced domains are then *propagated* to other constraints, whose filtering mechanisms reduce them further.[2]

Decision diagrams have been proposed as a data structure for certain filtering algorithms. For example, they are used in [70, 90, 107] for constraints defined on *set variables*, whose domains are sets of sets. They have also been used in [44, 45] to help filter "table" constraints, which are defined by an explicit list of allowed tuples for a set of variables.

It is important to note that in this research, decision diagrams help to filter domains for one constraint at a time, while information is conveyed to other constraints in the standard manner through individual variable domains (i.e., through a *domain store*). However, decision diagrams can be used for propagation as well, as initially pointed out by Andersen, Hadžić, Hooker and Tiedemann [4]. Their approach, and the one emphasized in this book, is to transmit information though a "relaxed" decision diagram rather than through a domain store, as discussed in the next section. Another approach is to conjoin MDDs associated with constraints that contain only a few variables in common, as later proposed by Hadžić, O'Mahony, O'Sullivan and Sellmann [87] for the market split problem. Either mechanism propagates information about inter-variable relationships, as well as about individual variables, and can therefore reduce the search significantly.

2.3.4 Relaxed Decision Diagrams

The concept of a *relaxed decision diagram* introduced by Andersen, Hadžić, Hooker and Tiedemann [4] plays a fundamental role in this book. This is a decision diagram that represents a superset of the feasible solutions and therefore provides a *discrete relaxation* of the problem, as contrasted with the continuous relaxations typically used in optimization. A key advantage of relaxed decision diagrams is that they can be much smaller than exact ones while still providing a useful relaxation, if they are properly constructed. In fact, one can control the size of a relaxed diagram by specifying an upper bound on the width as the diagram is built. A larger bound results in a diagram that more closely represents the original problem.

[2] Chapter 9 describes the filtering process in more detail.

Andersen et al. originally proposed relaxed decision diagrams as an enhanced propagation medium for constraint programming, as noted above. They developed two propagation mechanisms: the removal of arcs that are not used by any solution, and *node refinement*, which introduces new nodes in order to represent the solution space more accurately. They implemented MDD-based propagation for a system of ALLDIFFERENT constraints (which is equivalent to the graph coloring problem) and showed experimentally that it can result in a solution that is order of magnitude faster than using the conventional domain store. MDD-based propagation for equality constraints was studied in [85]. Following this, generic methods were developed in [84, 94] for systematically compiling relaxed decision diagrams in a top-down fashion. The details will described in the remainder of the book, but the fundamental idea is to construct the diagram in an incremental fashion, associating *state* information with the nodes of the diagram to indicate how new nodes and arcs should be created.

This kind of MDD-based propagation can be added to an existing constraint programming solver by treating the relaxed decision diagram as a new global constraint. Ciré and van Hoeve [49] implemented this approach and applied it to sequencing problems, resulting in substantial improvements over state-of-the-art constraint programming, and closing several open problem instances.

Relaxed decision diagrams can also provide optimization bounds, because the shortest top-to-bottom path length in a diagram is a lower bound on the optimal value (of a minimization problem). This idea was explored by Bergman, Ciré, van Hoeve and Hooker in [25, 28], who used state information to build relaxed decision diagrams for the set covering and stable set problems. They showed that relaxed diagrams can yield tighter bounds, in less time, than the full cutting plane resources of commercial integer programming software. Their technique would become a key component of a general-purpose optimization method based on decision diagrams.

2.3.5 A General-Purpose Solver

Several elements converged to produce a general-purpose discrete optimization method that is based entirely on decision diagrams. One is the top-down compilation method for generating a relaxed decision diagram already discussed. Another is a compilation method for *restricted* decision diagrams, which are important for obtaining good feasible solutions (i.e., as a *primal heuristic*). A restricted diagram is

one that represents a proper subset of feasible solutions. Bergman, Ciré, van Hoeve and Yunes [27] showed that restricted diagrams are competitive with the primal heuristics in state-of-the-art solvers when applied to set covering and set packing problems.

A third element is the connection between decision diagrams and dynamic programming, studied by Hooker in [97]. A *weighted* decision diagram, which is one in which costs are associated with the arcs, can be viewed as the state transition graph for a dynamic programming model. This means that problems are most naturally formulated for an MDD-based solver as dynamic programming models. The state variables in the model are those used in the top-down compilation of relaxed and restricted diagrams.

One advantage of dynamic programming models is that they allow for state-dependent costs, affording a great deal of flexibility in the choice of objective function. A given state-dependent cost function can be represented in multiple ways by assigning costs to arcs of an MDD, but it is shown in [97] that if the cost assignment is "canonical," there is a unique reduced weighted diagram for the problem. This generalizes the uniqueness theorem for classical reduced decision diagrams. A similar result is proved by Sanner and McAllester [138] for affine algebraic decision diagrams. The use of canonical costs can reduce the size of a weighted decision diagram dramatically, as is shown in [97] for a textbook inventory management problem.

A solver based on these elements is described in [26]. It uses a branch-and-bound algorithm in which decision diagrams play the role of the linear programming relaxation in traditional integer programming methods. The solver also uses a novel search scheme that branches on nodes of a relaxed decision diagram rather than on variables. It proved to be competitive with or superior to a state-of-the-art integer programming solver on stable set, maximum cut, and maximum 2-SAT problems, even though integer programming technology has improved by orders of magnitude over decades of solver development.

The use of relaxed decision diagrams in the solver has a superficial resemblance to *state space relaxation* in dynamic programming, an idea introduced by Christofides, Mingozzi and Toth [47]. However, there are fundamental differences. Most importantly, the problem is solved exactly by a branch-and-bound search rather than approximately by enumerating states. In addition, the relaxation is created by splitting or merging nodes in a decision diagram (state transition graph) rather than mapping the state space into a smaller space. It is tightened by filtering

techniques from constraint programming, and it is constructed dynamically as the decision diagram is built, rather than by defining a mapping a priori. Finally, the MDD-based relaxation uses the same state variables as the exact formulation, which allows the relaxed decision diagram to serve as a branching framework for finding an exact solution of the problem.

2.3.6 Markov Decision Processes

Decision diagrams have also played an auxiliary role in the solution of planning problems that arise in the artificial intelligence (AI) literature. These problems are often modeled as stochastic dynamic programming problems, because a given action or control can result in any one of several state transitions, each with a given probability. Nearly all the attention in AI has been focused on *Markov decision processes*, a special case of stochastic dynamic programming in which the state space and choice of actions are the same in each period or stage. A Markov decision process can also be *partially observable*, meaning that one cannot observe the current state directly but can observe only a noisy signal that indicates that the system could be in one of several possible states, each with a known probability.

The solution of stochastic dynamic programming models is complicated not only by the large state spaces that characterize deterministic models, but by the added burden of calculating expected immediate costs and costs-to-go that depend on probabilistic outcomes.[3] A natural strategy is to simplify and/or approximate the cost functions, an option that has been explored for many years in the optimization world under the name *approximate dynamic programming* (see [129] for a survey). The AI community has devised a similar strategy. The most obvious approximation technique is *state aggregation*, which groups states into sets and lets a single state represent each set. A popular form of aggregation in AI is “abstraction,” in which states are implicitly grouped by ignoring some of the problem variables.

This is where decision diagrams enter the picture. The cost functions are simplified or approximated by representing them with weighted decision diagrams, or rather *algebraic decision diagrams* (ADDs), which are a special case of weighted decision diagrams in which costs are attached to terminal nodes. One well-known

[3] The expected immediate cost of an action in a given state is the expected cost of taking that action in that state. The expected cost-to-go is the expected total cost of taking that action and following an optimal policy thereafter.

approach [95] uses ADDs as an abstraction technique to simplify the immediate cost functions in fully observable Markov decision processes. Some related techniques are developed in [63, 143].

Relaxation is occasionally used in these methods, but it is very different from the type of relaxation described above. Perhaps the closest analog appears in [146], which uses ADDs to represent a relaxation of the cost-to-go function, thereby providing a valid bound on the cost. Specifically, it attaches cost intervals to leaf nodes of an ADD that represents the cost function. The ADD is reduced by merging some leaf nodes and taking the union of the associated intervals. This does not create a relaxation of the entire recursion, as does node merger as employed in this book, but only relaxes the cost-to-go in an individual stage of the recursion. The result is a relaxation that embodies less information about the interaction of stages.

On the other hand, the methods we present here do not accommodate *stochastic* dynamic programming. All state transitions are assumed to be deterministic. It is straightforward to define a stochastic decision diagram, in analogy with the transition graph in stochastic dynamic programming, but it is less obvious how to *relax* a stochastic decision diagram by node merger or other techniques. This poses an interesting research issue that is currently under study.

Chapter 3
Exact Decision Diagrams

Abstract In this chapter we introduce a modeling framework based on dynamic programming to compile exact decision diagrams. We describe how dynamic programming models can be used in a top-down compilation method to construct exact decision diagrams. We also present an alternative compilation method based on constraint separation. We illustrate our framework on a number of classical combinatorial optimization problems: maximum independent set, set covering, set packing, single machine scheduling, maximum cut, and maximum 2-satisfiability.

3.1 Introduction

In this chapter we introduce a modeling framework based on dynamic programming (DP) to compile *exact* decision diagrams, i.e., decision diagrams that exactly represent the feasible solutions to a discrete optimization problem. We show two compilation techniques that can exploit this framework: the *top-down compilation* method and an alternative method based on *constraint separation*. Top-down compilation exploits a complete recursive description of the problem, and it is directly derived from the DP model of the problem. Constraint separation, in turn, is more suitable to problems that are more naturally written as a composition of models, each representing a particular substructure of the constraint set.

The chapter is organized as follows: In Section 3.2 we introduce the basic concepts of exact decision diagrams and the notation to be used throughout this book. Section 3.4 presents the modeling framework and the top-down compilation procedure, which are exemplified in a number of classical optimization problems.

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_3

Table 3.1 Data for a small knapsack problem.

Item	*Profit*	*Weight*
1	8	3
2	7	3
3	6	4
4	14	6
Capacity: 6		

Section 3.11 presents the constraint by separation method. Finally, Section 3.12 shows the validity of some key DP formulations used throughout this chapter.

3.2 Basic Definitions

In this book we focus on *discrete optimization problems* of the form

$$\begin{aligned} \max \ & f(x) \\ & C_i(x), \ \ i = 1,\ldots,m \\ & x \in D, \end{aligned} \qquad (\mathscr{P})$$

where $x = (x_1,\ldots,x_n)$ is a tuple of n decision variables, f is a real-valued function over x, $C_1,\ldots,C_m$ is a set of m constraints, and $D = D(x_1) \times \cdots \times D(x_n)$ is the Cartesian product of the *domains* of the variables, i.e., $x_j \in D(x_j)$ for each j. We assume here that $D(x_j)$ is finite for all x_j. A constraint $C_i(x)$ states an arbitrary relation between two or more variables, and it is *satisfied* by x if the relation is observed and *violated* otherwise. A *solution* to $\mathscr{P}$ is any $x \in D$, and a *feasible solution* to $\mathscr{P}$ is any solution that satisfies all constraints $C_i(x)$. The set of feasible solutions of $\mathscr{P}$ is denoted by $\mathrm{Sol}(\mathscr{P})$. A feasible solution x^* is *optimal* for $\mathscr{P}$ if $f(x^*) \geq f(x)$ for all $x \in \mathrm{Sol}(\mathscr{P})$. We denote by $z^* = f(x^*)$ the optimal solution value of $\mathscr{P}$.

A classical example of a discrete optimization problem is the *0/1 knapsack problem*. Given n items, each associated with a weight and a profit, we wish to select a subset of the items so as to maximize the sum of profits while keeping the total weight within a specified capacity. For example, Table 3.1 depicts a small knapsack problem with four items and a knapsack capacity of 6. This instance can be written as the following discrete optimization problem:

$$\begin{aligned} \max\ & 8x_1 + 7x_2 + 6x_3 + 14x_4 \\ & 3x_1 + 3x_2 + 4x_3 + 6x_4 \leq 6 \\ & x_j \in \{0,1\},\ \ j = 1,\ldots,4. \end{aligned} \tag{3.1}$$

In the formulation above, we define a variable x_j for each item j with binary domain $D(x_j) = \{0,1\}$ indicating whether item j is selected ($x_j = 1$) or not ($x_j = 0$). The objective function is the total profit of the selected items, and there is a single linear constraint enforcing the weight capacity. The set of feasible solutions is $\mathrm{Sol}(\mathscr{P}) = \{(0,0,0,0),(1,0,0,0),(0,1,0,0),(0,0,1,0),(0,0,0,1),(1,1,0,0)\}$. The optimal solution is $x^* = (1,1,0,0)$ and has a value of $z^* = 15$.

3.3 Basic Concepts of Decision Diagrams

For the purposes of this book, a *decision diagram* (DD) is a graphical structure that encodes a set of solutions to a discrete optimization problem $\mathscr{P}$. Formally, $B = (U,A,d)$ is a layered directed acyclic multigraph with node set U, arc set A, and arc labels d. The node set U is partitioned into layers $L_1,\ldots,L_{n+1}$, where layers L_1 and L_{n+1} consist of single nodes, the root node $\mathbf{r}$ and the terminal node $\mathbf{t}$, respectively. Each arc $a \in A$ is directed from a node in some L_j to a node in L_{j+1} and has a label $d(a) \in D(x_j)$ that represents the assignment of value $d(a)$ to variable x_j. Thus, every arc-specified path $p = (a^{(1)},\ldots,a^{(n)})$ from $\mathbf{r}$ to $\mathbf{t}$ encodes an assignment to the variables $x_1,\ldots,x_n$, namely $x_j = d(a^{(j)})$ for $j = 1,\ldots,n$. We denote this assignment by x^p. The set of $\mathbf{r}$ to $\mathbf{t}$ paths of B represents the set of assignments $\mathrm{Sol}(B)$.

Figure 3.1 depicts a decision diagram B for the knapsack problem (3.1). The diagram is composed of five layers, where the first four layers correspond to variables $x_1,\ldots,x_4$, respectively. Every arc a in B represents either a value of 0, depicted as a dashed arc in the figure, or a value of 1, depicted as a solid arc; e.g., $d((u_1,v_1)) = 0$. In particular, this DD encodes exactly the set of feasible solutions to the knapsack problem (3.1). For example, the path $p = (\mathbf{r},u_1,v_2,w_2,\mathbf{t})$ represents the assignment $x^p = (0,1,0,0)$.

The *width* $|L_j|$ of layer L_j is the number of nodes in the layer, and the *width* of a DD is $\max_j\{|L_j|\}$. The size $|B|$ of a DD B is given by its number of nodes. For instance, the width of B in Fig. 3.1 is 2 and $|B| = 8$. No two arcs leaving the same node have the same label, which means that every node has a maximum out-degree of $|D(x_j)|$. If all variables are binaries, then the DD is a *binary decision diagram*

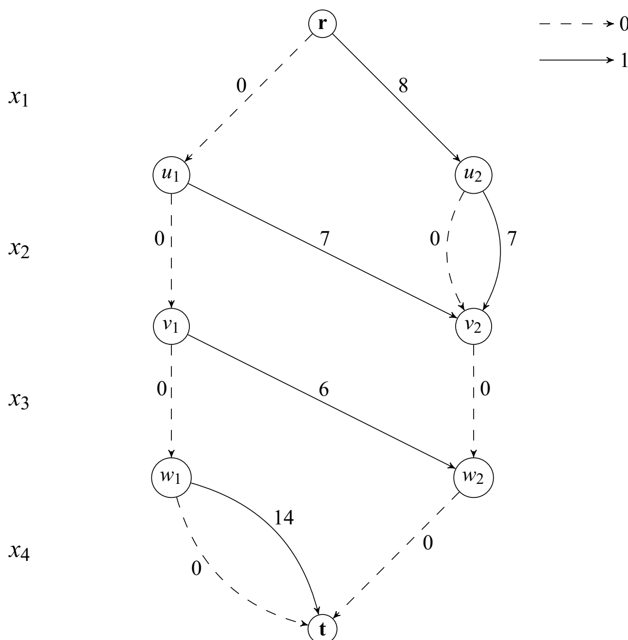


Fig. 3.1 Exact BDD for the knapsack instance of Table 3.1. Dashed and solid arcs represent arc labels 0 and 1, respectively. The numbers on the arcs indicate their length.

(BDD), which has been the subject of the majority of studies in the area due to the applications in Boolean logic [110, 99, 37]. On the other hand, a *multivalued decision diagram* (MDD) allows out-degrees higher than 2 and therefore encodes values of general finite-domain variables.

Because we are interested in optimization, we focus on *weighted* DDs, in which each arc a has an associated *length* $v(a)$. The length of a directed path $p = \left(a^{(1)}, \ldots, a^{(k)}\right)$ rooted at $\mathbf{r}$ corresponds to $v(p) = \sum_{j=1}^{k} v(a^{(j)})$. A weighted DD B represents an optimization problem $\mathscr{P}$ in a straightforward way. Namely, B is an *exact decision diagram representation* of $\mathscr{P}$ if the r–t paths in B encode precisely the feasible solutions of $\mathscr{P}$, and the length of a path is the objective function value of the corresponding solution. More formally, we say that B is *exact* for $\mathscr{P}$ when

$$\mathrm{Sol}(\mathscr{P}) = \mathrm{Sol}(B) \tag{3.2}$$

$$f(x^p) = v(p), \text{ for all } r\text{–}t \text{ paths } p \text{ in } B. \tag{3.3}$$

In Fig. 3.1 the length $v(a)$ is represented by a number on each arc a. One can verify that the BDD B depicted in this figure satisfies both conditions (3.2) and (3.3)

for the knapsack problem (3.1). For example, the path $p = (\mathbf{r}, u_1, v_2, w_2, \mathbf{t})$ has a length $v(p) = 3$, which coincides with $f(x^p) = f((0,1,0,0)) = 3$.

An exact DD reduces discrete optimization to a longest-path problem on a directed acyclic graph. If p is a longest path in a DD B that is exact for $\mathscr{P}$, then x^p is an optimal solution of $\mathscr{P}$, and its length $v(p)$ is the optimal value $z^*(\mathscr{P}) = f(x^p)$ of $\mathscr{P}$. For Fig. 3.1, the longest path is given by the path p^* with length $v(p^*) = 15$ that crosses nodes $(\mathbf{r}, u_2, v_2, w_2, \mathbf{t})$, representing the optimal solution $x^{p^*} = (1,1,0,0)$.

It is common in the DD literature to allow various types of *long arcs* that skip one or more layers [37, 116]. Long arcs can improve efficiency because they represent multiple partial assignments with a single arc, but to simplify exposition, we will suppose with minimal loss of generality that there are no long arcs throughout this book. DDs also typically have two terminal nodes, corresponding to true and false, but for our purposes only a true node is required as the terminal for feasible paths.

Given a DD B, two nodes belonging to the same layer L_j are *equivalent* when the paths from each to the terminal $\mathbf{t}$ are the same; i.e., they correspond to the same set of assignments to $(x_j, \ldots, x_n)$, which implies they are redundant in the representation. A *reduced DD* is such that no two nodes of a layer are equivalent, as in the case of the DD in Fig. 3.1. For a given ordering of the variables over the diagram layers, there exists a unique (*canonical*) reduced DD which has the smallest width across DDs with that ordering. A DD can be reduced in linear time on the number of arcs and nodes of the graph [37, 157].

3.4 Compiling Exact Decision Diagrams

We now present a generic framework for compiling an exact decision diagram encoding the solutions of a discrete optimization problem $\mathscr{P}$. The framework requires $\mathscr{P}$ to be written as a dynamic programming (DP) model and extracts a decision diagram from the resulting state transition graph. We first describe the elements of a dynamic programming model, then outline the details of our framework, and finally show DD examples on different problem classes.

3.4.1 Dynamic Programming

Dynamic programming (DP) is a *recursive* optimization method in which a discrete optimization problem $\mathscr{P}$ is formulated in terms of *states* as opposed to variable assignments. A DP model is solved in stages, each representing the transition from a particular state of the system to the next until a final (or *terminal*) state is reached. Each transition is governed by a variable of the problem and incurs a value. The optimal solution to the problem corresponds to a maximum-value set of transitions from a given *root state* to the terminal state.

To illustrate the main ingredients of a DP model, recall the knapsack problem (3.1). Suppose the weights and profits of item j are w_j and p_j, respectively, and that the knapsack capacity is U. We consider a DP formulation where each state represents the total weight of the selected items up to that stage. Namely, the state s^j at stage j represents the weight considering that items $1,2,\ldots,j-1$ have already been considered for selection or not.

In the initial stage no items have been considered thus far, hence the root state is such that $s^1 = 0$. The transition from stage j to stage $j+1$ depends on the variable x_j (also denoted by *control* in DP jargon). In stage j, if the control x_j selects item j (i.e., $x_j = 1$), then the total weight increases by w_j and we transition to state $s^{j+1} = s^j + w_j$. Otherwise, the total weight remains the same and we transition to $s^{j+1} = s^j$. Thus, $s^{j+1} = s^j + w_j x_j$. Since we are only interested in feasible solutions, the final state s^{n+1}, denoted by *terminal state*, must satisfy $s^{n+1} \le U$.

Finally, the objective function is represented by profits incurred by each transition. For the knapsack, we assign a cost of 0 if $x_j = 0$, and p_j otherwise. We wish to find a set of transitions that lead us from the root state to the terminal state with maximum transition profit. By representing the reached states as *state variables* in an optimization model, the resulting DP model for the 0/1 knapsack problem is given as follows:

$$
\begin{aligned}
\max \quad & \sum_{j=1}^{n} p_j x_j \\
& s^{j+1} = s^j + w_j x_j, \;\; j = 1,\ldots,n \\
& s^1 = 0, \; s^{n+1} \le U \\
& x_j \in \{0,1\}, \;\; j = 1,\ldots,n.
\end{aligned}
\tag{3.4}
$$

One can verify that any valid solution (x,s) to the DP model above leads to an assignment x that is feasible to the knapsack model (3.1). Conversely, any feasible assignment x to model (3.1) has a unique completion (s,x) that is feasible to the DP model above, thus both models are equivalent. Notice also that the states are *Markovian*; i.e., the state s^{j+1} only depends on the control x_j and the previous state s^j, which is a fundamental property of DP models.

The main components of a DP model are the states, the way in which the controls govern the transitions, and finally the costs of each transition. To specify this formally, a DP model for a given problem $\mathscr{P}$ with n variables having domains $D(x_1),\dots,D(x_n)$ must, in general, consist of the following four elements:

1. A *state space* S with a *root state* $\hat{r}$ and a countable set of *terminal states* $\hat{t}_1,\hat{t}_2,\dots,\hat{t}_k$. To facilitate notation, we also consider an *infeasible state* $\hat{0}$ that leads to infeasible solutions to $\mathscr{P}$. The state space is partitioned into sets for each of the $n+1$ stages; i.e., S is the union of the sets $S_1,\dots,S_{n+1}$, where $S_1 = \{\hat{r}\}$, $S_{n+1} = \{\hat{t}_1,\dots,\hat{t}_k,\hat{0}\}$, and $\hat{0} \in S_j$ for $j = 2,\dots,n$.
2. *Transition functions* t_j representing how the controls govern the transition between states; i.e., $t_j : S_j \times D_j \to S_{j+1}$ for $j = 1,\dots,n$. Also, a transition from an infeasible state always leads to an infeasible state as well, regardless of the control value: $t_j(\hat{0},d) = \hat{0}$ for any $d \in D_j$.
3. *Transition cost functions* $h_j : S \times D_j \to \mathbb{R}$ for $j = 1,\dots,n$.
4. To account for objective function constants, we also consider a *root value* v_r, which is a constant that will be added to the transition costs directed out of the root state.

The DP formulation has variables $(s,x) = (s^1,\dots,s^{n+1},x_1,\dots,x_n)$ and is written

$$
\begin{aligned}
\min \quad & \hat{f}(s,x) = \sum_{j=1}^{n} h_j(s^j,x_j) \\
& s^{j+1} = t_j(s^j,x_j), \quad \text{for all } x_j \in D_j,\ j = 1,\dots,n \\
& s^j \in S_j, \quad j = 1,\dots,n+1.
\end{aligned}
\tag{3.5}
$$

The formulation (3.5) is *valid* for $\mathscr{P}$ if, for every $x \in D$, there is an $s \in S$ such that (s,x) is feasible in (3.5) and

$$
s^{n+1} = \hat{t} \text{ and } \hat{f}(s,x) = f(x), \text{ if } x \text{ is feasible for } \mathscr{P} \tag{3.6}
$$

$$
s^{n+1} = \hat{0}, \text{ if } x \text{ is infeasible for } \mathscr{P}. \tag{3.7}
$$

Algorithm 1 Exact DD Top-Down Compilation

1: Create node $r = \hat{r}$ and let $L_1 = \{r\}$
2: **for** $j = 1$ to n **do**
3: let $L_{j+1} = \emptyset$
4: **for all** $u \in L_j$ and $d \in D_j$ **do**
5: **if** $t_j(u,d) \neq \hat{0}$ **then**
6: let $u' = t_j(u,d)$, add u' to L_{j+1}, and set $b_d(u) = u'$, $v(u,u') = h_j(u,u')$
7: Merge nodes in L_{n+1} into terminal node $\mathbf{t}$

3.4.2 Top-Down Compilation

The construction of an exact weighted decision diagram from a DP formulation is straightforward in principle. For a DD B, let $b_v(u)$ denote the node at the opposite end of an arc leaving node u with value v (if it exists). The compilation procedure is stated as Algorithm 1 and in essence builds the BDD layer by layer starting at the root node, as follows: Begin with the root node $\mathbf{r}$ in layer 1, which corresponds to the root state $\hat{r}$. Proceed recursively, creating a node for each feasible state that can be reached from $\mathbf{r}$. Thus, having constructed layer j, let L_{j+1} contain nodes corresponding to all *distinct* feasible states to which one can transition from states represented in L_j. Then add an arc from layer j to layer $j+1$ for each such transition, with length equal to the transition cost. At the last stage, identify all terminal states $\hat{t}_1, \dots, \hat{t}_k$ to a terminal node $\mathbf{t}$. Because distinct nodes always have distinct states, the algorithm identifies each node with the state associated with that node.

Figure 3.2 depicts three consecutive iterations of Algorithm 1 for the 0/1 knapsack problem (3.1). The DP states from model (3.4) are represented as grey boxes next to the nodes that identify them. In the first iteration, presented in Fig. 3.2(a), layer L_1 is built with the root node $\mathbf{r}$ having state 0, and layer L_2 is built with nodes u_1 and u_2 having states 0 and 3, respectively. In the second iteration, layer L_3 is built with three nodes, as Fig. 3.2(b) shows. Since the arc with label 1 leaving node u_1 and the arc with label 0 leaving node u_1 transition to the same state, their endpoints are directed to the same node at layer L_3. Fig. 3.2(c) depicts one more iteration for the construction of layer L_4. Note that nodes with states 7 and 10 can be removed, since they violate the knapsack capacity of 6.

Algorithm 1 assumes that the controls $x_1, \dots, x_n$ are ordered according to the DP model input. Nevertheless, as studied in [18], it is often possible to reorder the controls and obtain DDs of drastically different sizes. In the context of decision diagrams for optimization, the orderings and the respective size of a DD are closely

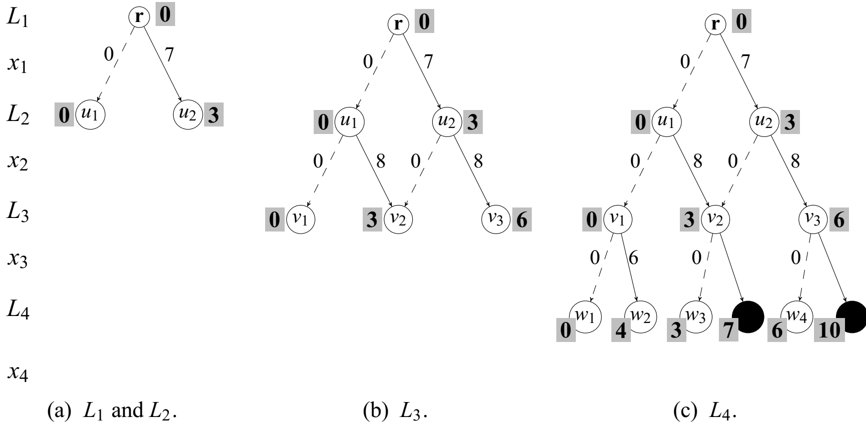


Fig. 3.2 Three consecutive iterations of Algorithm 1 for the 0/1 knapsack problem (3.1). Grey boxes correspond to the DP states in model (3.4), and black-filled nodes indicate infeasible nodes.

related to the combinatorial structure of the problem that the DD represents. We present a study case on this relationship in Chapter 7.

The outlined DD construction procedure can also be perceived as a particular representation of the *state-graph* of the DP formulation [97]. Suppose that (3.5) is valid for problem $\mathscr{P}$, and consider the state-transition graph for (3.5). Omit all occurrences of the infeasible state $\hat{0}$, and let each remaining arc from state s^j to state $t_j(s^j, x_j)$ have length equal to the transition cost $h_j(s^j, x_j)$. The resulting multigraph B_{DP} is an exact DD for $\mathscr{P}$, because paths from state r to state t in B_{DP} correspond precisely to feasible solutions of (3.5), and the objective function value of the corresponding solution is the path length. A thorough discussion of this relationship is discussed in Chapter 8

Finally, we remark in passing that the resulting DD is not necessarily reduced. Note that, in the DD examples of Fig. 3.2, nodes v_2 and v_3 in layer L_3 are equivalent according to the exact BDD in Fig. 3.1, but they are not merged in the top-down compilation procedure. In general, identifying if two nodes are equivalent is NP-hard (which is the case of the knapsack problem), though this can be done efficiently for certain classes of combinatorial problems such as the maximum independent set (Section 3.5) and set covering (Section 3.9). Moreover, although reduced DDs play a key role in circuit verification and some other applications, they can be unsuitable for optimization, because the arc lengths from equivalent nodes may differ. This will be the case, e.g., for maximum cut problems described in Section 3.9.

In the next section we exemplify the DP formulation and the diagram construction for different problem classes in optimization. To facilitate reading, proofs certifying the validity of the formulations are shown in Section 3.12.

3.5 Maximum Independent Set Problem

Given a graph $G = (V, E)$ with an arbitrarily ordered vertex set $V = \{1, 2, \ldots, n\}$, an *independent set* I is a subset $I \subseteq V$ such that no two vertices in I are connected by an edge in E. If we associate weights $w_j \geq 0$ with each vertex $j \in V$, the maximum independent set problem (MISP) asks for a maximum-weight independent set of G. For example, in the graph depicted in Fig. 3.3, the maximum weighted independent set is $I = \{2, 5\}$ and has a value of 13. The MISP (which is equivalent to the maximum clique problem) has found applications in many areas, including data mining [61], bioinformatics [59], and social network analysis [16].

The MISP can be formulated as the following discrete optimization problem:

$$\begin{aligned} \max \; & \sum_{j=1}^{n} w_j x_j \\ & x_i + x_j \leq 1, \;\; \text{for all } (i, j) \in E \\ & x_j \in \{0, 1\}, \;\; \text{for all } j \in V. \end{aligned} \tag{3.8}$$

In the formulation above, we define a variable x_j for each vertex $j \in V$ with binary domain $D(x_j) = \{0, 1\}$, indicating whether vertex j is selected ($x_j = 1$) or not ($x_j = 0$). The objective function is the weight of the independent set, $f(x) = \sum_{j=1}^{n} w_j x_j$, and the constraint set prevents two vertices connected by an edge from being selected simultaneously. For the graph in Fig. 3.3, the corresponding model is such that the optimal solution is $x^* = (0, 1, 0, 0, 1)$ and the optimal solution value is $z^* = 13$. Moreover, $\mathrm{Sol}(\mathscr{P})$ is defined by the vectors x that represent the family of independent sets $V \cup \{\{1, 4\}, \{1, 5\}, \{2, 5\}, \{3, 5\}\}$.

Figure 3.4 shows an exact weighted BDD B for the MISP problem defined over the graph G in Fig. 3.3. Any r–t path in B represents a variable assignment that corresponds to a feasible independent set of G; conversely, all independent sets are represented by some r–t path in B, hence $\mathrm{Sol}(\mathscr{P}) = \mathrm{Sol}(B)$. The size of B is $|B| = 11$, and its width is 3, which is the width of layer L_3. Finally, notice that the length of each path p corresponds exactly to the weight of the independent set

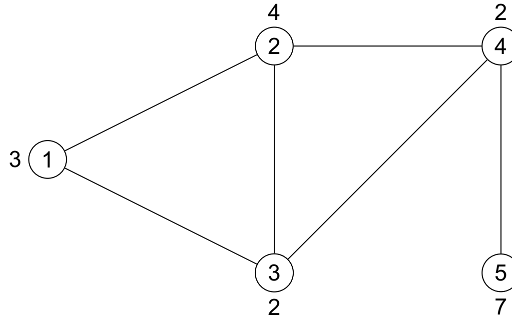


Fig. 3.3 Example of a graph with vertex weights for the MISP. Vertices are assumed to be labeled arbitrarily, and the number alongside each circle indicates the vertex weight.

represented by x^p. In particular, the longest path in B has a value of 13 and yields the assignment $x^* = (0,1,0,0,1)$, which is the optimum solution to the problem.

To formulate a DP model for the MISP, we introduce a state space where in stage j we decide if vertex j will be added to a partial independent set, considering that we have already decided whether vertices $1,2,\ldots,j-1$ are in this partial independent set or not. In particular, each state s^j in our formulation represents *the set of vertices that still can be added* to the partial independent set we have constructed up to stage j. In the first stage of the system, no independent set has been considered so far, thus the root state is $\hat{r} = V$, i.e., all vertices are eligible to be added. The terminal state is

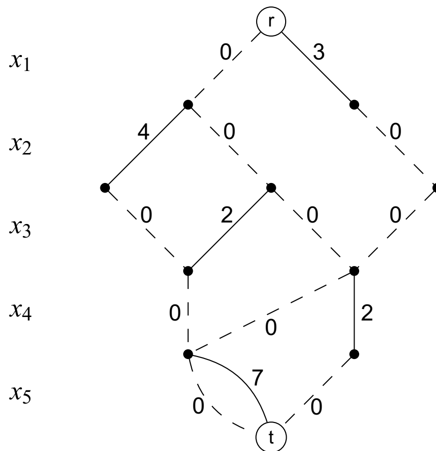


Fig. 3.4 Exact BDD for the MISP on the graph in Fig. 3.3. Dashed and solid arcs represent labels 0 and 1, respectively.

when no more vertices can be considered, $\hat{0} = \emptyset$. Finally, a state s^j in the j-th stage is such that $s^j \subseteq \{j, j+1, \ldots, n\}$.

Given a state s^j in stage j of the MISP, the assignment of the control x_j defines the next state s^{j+1}. If $x_j = 0$, then by definition vertex j *is not* added to the independent set thus far, and hence $s^{j+1} = s_j \setminus \{j\}$. If $x_j = 1$, we add vertex j to the existing independent set constructed up to stage j, but now we are unable to add any of the adjacent vertices of j, $N(j) = \{j' \mid (j, j') \in E\}$, to our current independent set. Thus the new eligibility vertex set is $s^{j+1} = s_j \setminus (N(j) \cup \{j\})$. Note that the transition triggered by $x_j = 1$ will lead to an infeasible independent set if $j \notin s_j$. Finally, we assign a transition cost of 0 if $x_j = 0$, and w_j otherwise. We wish to find a set of transitions that lead us from the root state to the terminal state with maximum cost.

Formally, the DP model of the MISP is composed of the following components:

- State spaces: $S_j = 2^{V_j}$ for $j = 2, \ldots, n$, $\hat{r} = V$, and $\hat{t} = \emptyset$
- Transition functions: $t_j(s^j, 0) = s^j \setminus \{j\}$, $\quad t_j(s^j, 1) = \begin{cases} s^j \setminus N(j) & , \text{ if } j \in s^j \\ \hat{0} & , \text{ if } j \notin s^j \end{cases}$
- Cost functions: $h_j(s^j, 0) = 0$, $h_j(s^j, 1) = w_j$
- A root value of 0

As an illustration, consider the MISP for the graph in Fig. 3.3. The states associated with nodes of B_{DP} are shown in Fig. 3.5. For example, node u_1 has state $\{2,3,4,5\}$, representing the vertex set $V \setminus \{1\}$. The state space described above yields a reduced DD [23], thus it is the smallest possible DD for a fixed ordering of variables over the layers.

3.6 Set Covering Problem

The *set covering problem* (SCP) is the binary program

$$\begin{aligned} & \min \ c^T x \\ & Ax \geq e \\ & x_j \in \{0,1\}, \ j = 1, \ldots, n, \end{aligned}$$

where c is an n-dimensional real-valued vector, A is a 0–1 $m \times n$ matrix, and e is the m-dimensional unit vector. Let $a_{i,j}$ be the element in the i-th row and j-th column of A, and define $A_j = \{i \mid a_{i,j} = 1\}$ for $j = 1, \ldots, n$. The SCP asks for a minimum-

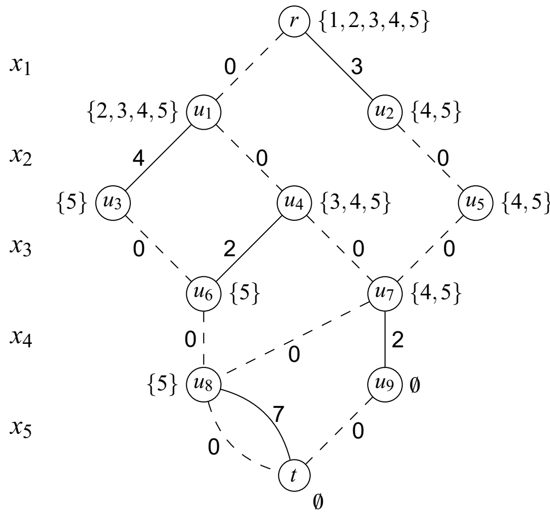


Fig. 3.5 Exact BDD with states for the MISP on the graph in Fig. 3.3.

cost subset $V \subseteq \{1,\ldots,n\}$ of the sets A_j such that, for all $i, a_{i,j} = 1$ for some $j \in V$, i.e., V *covers* $\{1,\ldots,m\}$. It is widely applied in practice, and it was one of the first combinatorial problems to be proved NP-complete [71].

We now formulate the SCP as a DP model. The state in a particular stage of our model indicates *the set of constraints that still need to be covered*. Namely, let C_i be the set of indices of the variables that participate in constraint i, $C_i = \{j \mid a_{i,j} = 1\}$, and let $\texttt{last}(C_i) = \max\{j \mid j \in C_i\}$ be the largest index of C_i. The components of the DP model are as follows:

- State spaces: In any stage, a state contains the set of constraints that still need to be covered: $S_j = 2^{\{1,\ldots,m\}} \cup \{\hat{0}\}$ for $j = 2,\ldots,n$. Initially, all constraints need to be satisfied, hence $\hat{r} = \{1,\ldots,m\}$. There is a single terminal state which indicates that all constraints are covered: $\hat{t} = \emptyset$.
- Transition functions: Consider a state s^j in stage j. If the control satisfies $x_j = 1$ then all constraints that variable x_j covers, $A_j = \{i \mid a_{i,j} = 1\} = \{i : j \in C_i\}$, can be removed from s^j. However, if $x_j = 0$, then the transition will lead to an infeasible state if there exists some i such that $\texttt{last}(C_i) = j$, since then constraint i will never be covered. Otherwise, the state remains the same. Thus:

$$t_j(s^j,1) = s^j \setminus A_j$$

$$t_j(s^j,0) = \begin{cases} s^j, \text{ if } \texttt{last}(C_i) > j \text{ for all } i \in s^j, \\ \hat{0}, \text{ otherwise.} \end{cases}$$

- Cost functions: $h_j(s^j, x_j) = -c_j x_j$.
- A root value of 0.

Consider the SCP problem

$$\begin{aligned} &\text{minimize } 2x_1 + x_2 + 4x_3 + 3x_4 + 4x_5 + 3x_6 \\ &\text{subject to } x_1 + x_2 + x_3 \geq 1 \\ &\qquad x_1 + x_4 + x_5 \geq 1 \\ &\qquad x_2 + x_4 + x_6 \geq 1 \\ &\qquad x_i \in \{0,1\}, \ i = 1,\ldots,6. \end{aligned} \tag{3.9}$$

Figure 3.6 shows an exact reduced BDD for this SCP instance where the nodes are labeled with their corresponding states. If outgoing 1-arcs (0-arcs) of nodes in layer j are assigned a cost of c_j (zero), a shortest r–t path corresponds to the solution $(1,1,0,0,0,0)$ with an optimal value of 3.

Different than the MISP, this particular DP model does not yield reduced DDs in general. An example is the set covering problem

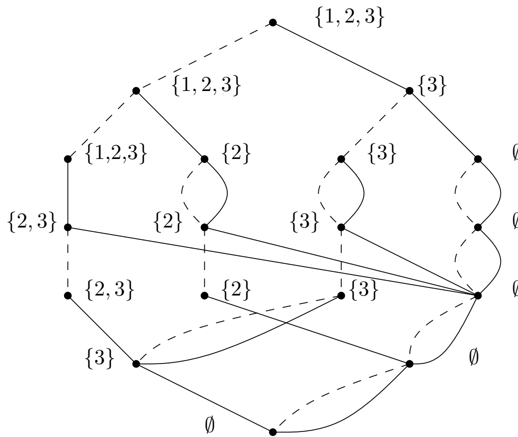


Fig. 3.6 Exact BDD for the SCP problem (3.9).

$$\begin{aligned} \text{minimize } & x_1 + x_2 + x_3 \\ \text{subject to } & x_1 + x_3 \geq 1 \\ & x_2 + x_3 \geq 1 \\ & x_1, x_2, x_3 \in \{0,1\} \end{aligned}$$

and the two partial solutions $x^1 = (1,0)$, $x^2 = (0,1)$. We have that the state reached by applying the first and second set of controls is $\{2\}$ and $\{1\}$, respectively. Thus, they would lead to different nodes in the resulting DD. However, both have the single feasible completion $\tilde{x} = (1)$.

There are several ways to modify the state function so that the resulting DD is reduced, as presented in [28]. This requires only polynomial time to compute per partial solution, but nonetheless at an additional computational cost.

3.7 Set Packing Problem

A problem closely related to the SCP, the *set packing problem* (SPP), is the binary program

$$\begin{aligned} \max \ & c^T x \\ & Ax \leq e \\ & x_j \in \{0,1\}, \ \ j = 1, \ldots, n, \end{aligned}$$

where c is an n-dimensional real-valued vector, A is a 0–1 $m \times n$ matrix, and e is the m-dimensional unit vector. Let $a_{i,j}$ be the element in the i-th row and j-th column of A, and define $A_j = \{i \mid a_{i,j} = 1\}$ for $j = 1, \ldots, n$. The SPP asks for the maximum-cost subset $V \subseteq \{1, \ldots, n\}$ of the sets A_j such that, for all $i, a_{i,j} = 1$ for at most one $j \in V$.

We now formulate the SPP as a DP model. The state in a particular stage of our model indicates *the set of constraints for which no variables have been assigned a 1 and could still be violated.* As in the SCP, let C_i be the set of indices of the variables that participate in constraint i, $C_i = \{j \mid a_{i,j} = 1\}$, and let $\texttt{last}(C_i) = \max\{j \mid j \in C_i\}$ be the largest index of C_i. The components of the DP model are as follows:

- State spaces: In any stage, a state contains the set of constraints for which no variables have been assigned a 1: $S_j = 2^{\{1,\ldots,m\}} \cup \{\hat{0}\}$ for $j = 2, \ldots, n$. Initially,

$\hat{r} = \{1,\dots,m\}$. There is a single terminal state $\hat{t} = \emptyset$, when no more constraints need to be considered.

- Transition functions: Consider a state s^j in stage j. By the definition of a state, the control $x_j = 1$ leads to the infeasible state $\hat{0}$ if there exists a constraint i that contains variable x_j ($i \in A_j$) and does *not* belong to s^j. If $x_j = 0$, then we can remove from s^j any constraints i for which $j = \texttt{last}(C_i)$, since these constraints will not be affected by the remaining controls. Thus:

$$t_j(s^j,0) = s^j \setminus \{i \mid \texttt{last}(C_i) = j\}$$

$$t_j(s^j,1) = \begin{cases} s^j \setminus \{i \mid j \in C_i\}, & \text{if } A_j \subseteq s^j, \\ \hat{0}, & \text{otherwise.} \end{cases}$$

- Cost functions: $h_j(s^j,x_j) = -c_j x_j$.
- A root value of 0.

Consider the SPP instance

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{6} x_i \\ \text{subject to } & x_1 + x_2 + x_3 \le 1 \\ & x_1 + x_4 + x_5 \le 1 \\ & x_2 + x_4 + x_6 \le 1 \\ & x_i \in \{0,1\}, \; i = 1,\dots,6. \end{aligned} \tag{3.10}$$

Figure 3.7 shows an exact reduced BDD for this SPP instance. The nodes are labeled with their corresponding states, and we assign arc costs 1/0 to each 1/0-arc. A longest r–t path, which can be computed by a shortest path on arc weights $c' = -c$ because the BDD is acyclic, corresponds to the solution $(0,0,1,0,1,1)$ and proves an optimal value of 3.

As in the case of the SCP, the above state function does not yield reduced DDs. The problem

$$\begin{aligned} \max \; & x_1 + x_2 + x_3 \\ & x_1 + x_3 \le 1 \\ & x_2 + x_3 \le 1 \\ & x_1, x_2, x_3 \in \{0,1\} \end{aligned}$$

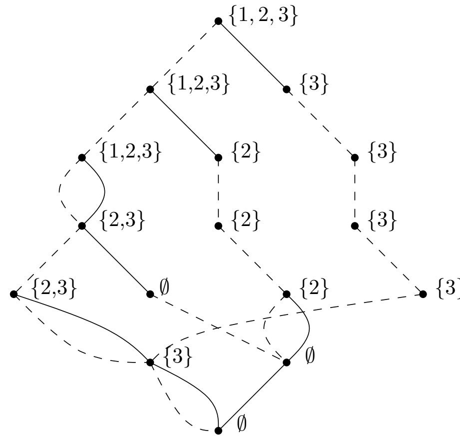


Fig. 3.7 Exact reduced BDD for the SPP instance (3.10).

has two partial solutions $x^1 = (1,0)$, $x^2 = (0,1)$. We have distinct states $\{2\}$ and $\{1\}$ reached by the controls x^1 and x^2, respectively, but both have the single feasible completion, $\tilde{x} = (0)$.

There are several ways to modify the state function above so that the DD construction algorithm outputs reduced decision diagrams. For example, one can reduce the SPP to an independent set problem and apply the state function defined in Section 3.5, which we demonstrate to have this property in Section 7.

3.8 Single-Machine Makespan Minimization

Let $\mathscr{J} = \{1,\ldots,n\}$ for any positive integer n. A *permutation* π of $\mathscr{J}$ is a complete ordering $(\pi_1, \pi_2, \ldots, \pi_n)$ of the elements of $\mathscr{J}$, where $\pi_i \in \mathscr{J}$ for all i and $\pi_i \neq \pi_j$ for all $i \neq j$. Combinatorial problems involving permutations are ubiquitous in optimization, especially in applications involving sequencing and scheduling.

For example, consider the following variant of a *single-machine makespan minimization problem* (MMP) [128]: Let $\mathscr{J}$ represent a set of n jobs that must be scheduled on a single machine. The machine can process at most one job at a time, and a job must be completely finished before starting the next job. With each job we associate a *position-dependent* processing time. Namely, let p_{ij} be the processing time of job j if it is the i-th job to be performed on the machine. We want to schedule jobs to minimize the total completion time, or *makespan*.

Table 3.2 Processing times of a single-machine makespan minimization problem. Rows and columns represent the job index and the position in the schedule, respectively.

	Position in Schedule		
Jobs	1	2	3
1	4	5	9
2	3	7	8
3	1	2	10

Table 3.2 depicts an instance of the MMP with three jobs. According to the given table, performing jobs 3, 2, and 1 in that order would result in a makespan of $1+7+9=17$. The minimum makespan is achieved by the permutation $(2,3,1)$ and has a value of $2+3+9=14$. Notice that the MMP presented here can be solved as a classical *matching problem* [122]. More complex position-dependent problems usually represent machine deterioration, and the literature on this topic is relatively recent [2].

To formulate the MMP as an optimization problem, we let x_i represent the i-th job to be processed on the machine. The MMP can be written as

$$
\begin{aligned}
\min \ & \sum_{i=1}^{n} p_{i,x_i} \\
& x_i \neq x_j, \quad i,j = 1,\ldots,n,\ i<j \qquad (3.11) \\
& x_i \in \{1,\ldots,n\},\ \ i = 1,\ldots,n.
\end{aligned}
$$

Constraints (3.11) indicate that variables $x_1,\ldots,x_n$ must assume pairwise distinct values; i.e., they define a permutation of $\mathscr{J}$. Hence, the set of feasible solutions to the MMP is the set of permutation vectors of $\mathscr{J}$. Note also that the objective function uses variables as indices, which will be shown to be naturally encoded in a DP model (and, consequently, easily represented in a MDD).

We now formulate the MMP as a DP model. The state in a particular stage of our model indicates *the jobs that were already performed on the machine*. The components of the DP model are as follows:

- State spaces: In a stage j, a state contains the $j-1$ jobs that were performed previously on the machine: $S_j = 2^{\{1,\ldots,n\}} \cup \{\hat{0}\}$ for $j=2,\ldots,n$. Initially, no jobs have been performed, hence $\hat{r}=\emptyset$. There is a single terminal state $\hat{t}=\{1,\ldots,n\}$, when all jobs have been completed.

- Transition functions: Consider a state s^j in stage j. By the definition of a state, the control $x_j = d$ for some $d \in \{1,\ldots,n\}$ simply indicates that job d will now be processed at stage j. The transition will lead to an infeasible state $\hat{0}$ if $d \in s^j$, because then job d has already been processed by the machine. Thus:

$$t_j(s^j,d) = \begin{cases} s^j \cup \{d\}, & \text{if } d \notin s^j, \\ \hat{0}, & \text{otherwise.} \end{cases}$$

- Cost functions: The transition cost corresponds to the processing time of the machine at that stage: $h_j(s^j,d) = -p_{j,d}$.
- A root value of 0.

Figure 3.8 depicts the MDD with node states for the MMP instance defined in Table 3.2. In particular, the path traversing nodes r, u_3, u_5, and t corresponds to processing jobs 3, 2, 1, in that order. This path has a length of 14, which is the optimal makespan of that instance.

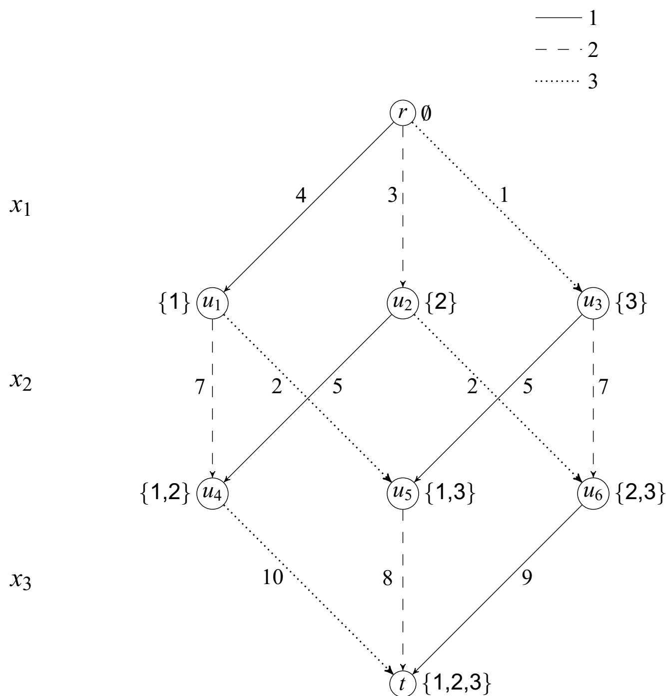


Fig. 3.8 Example of an MDD for the minimum makespan problem in Table 3.2. Solid, dashed, and dotted arcs represent labels 1, 2, and 3, respectively.

Existing works focus on representation issues of the set of permutation vectors (e.g., [141, 156, 13]). DD representations of permutations have also been suggested in the literature [117]. The DP model presented here yields the same DD as in [84]. It will be again the subject of our discussion in Chapter 11, where we discuss how to minimize different scheduling objective functions over the same decision diagram representation.

3.9 Maximum Cut Problem

Given a graph $G = (V,E)$ with vertex set $V = \{1,\dots,n\}$, a *cut* (S,T) is a partition of the vertices in V. We say that an edge crosses the cut if its endpoints are on opposite sides of the cut. Given edge weights, the value $v(S,T)$ of a cut is the sum of the weights of the edges crossing the cut. The *maximum cut problem* (MCP) is the problem of finding a cut of maximum value. The MCP has been applied to very-large-scale integration design, statistical physics, and other problems [88, 64].

To formulate the MCP as a binary optimization problem, let x_j indicate the set (S or T) in which vertex j is placed, so that $D_j = \{\mathrm{S},\mathrm{T}\}$. Using the notation $S(x) = \{j \mid x_j = \mathrm{S}\}$ and $T(x) = \{j \mid x_j = \mathrm{T}\}$, the objective function is $f(x) = v(S(x),T(x))$. Since any partition is feasible, $\mathscr{C} = \emptyset$. Thus the MCP can be written as

$$\begin{aligned} \max \;\; & v(S(x),T(x)) \\ & x_j \in \{S,T\}, \;\; j = 1,\dots,n. \end{aligned} \tag{3.12}$$

Consider the graph G depicted in Fig. 3.9. The optimal solution of the maximum cut problem defined over G is the cut $(S,T) = (\{1,2,4\},\{3\})$ and has a length of 4, which is the sum of the weights from edges $(1,3)$, $(2,3)$, and $(3,4)$. In our model this corresponds to the solution $x^* = (\mathrm{S},\mathrm{S},\mathrm{T},\mathrm{S})$ with $v(S(x^*),T(x^*)) = 4$.

We now formulate a DP model for the MCP. Let $G = (V,E)$ be an edge-weighted graph, which we can assume (without loss of generality) to be complete, because missing edges can be included with weight 0. A natural state variable s^j would be the set of vertices already placed in S, as this is sufficient to determine the transition cost of the next choice. However, we will be interested in merging nodes that lead to similar objective function values. We therefore let the state indicate, for vertex $j,\dots,n$, the net marginal benefit of placing that vertex in T, given previous choices. We will show that this is sufficient information to construct a DP recursion.

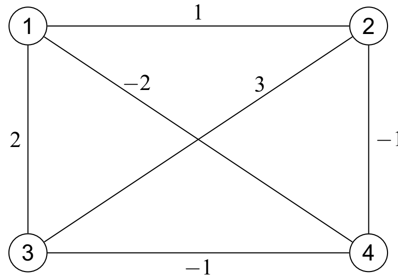


Fig. 3.9 Graph with edge weights for the MCP.

Formally, we specify the DP formulation as follows: As before, the control variable is $x_j \in \{\mathrm{S},\mathrm{T}\}$, indicating in which set vertex j is placed, and we set $x_1 = \mathrm{S}$ without loss of generality. We will use the notation $(\alpha)^+ = \max\{\alpha,0\}$ and $(\alpha)^- = \min\{\alpha,0\}$.

- State spaces: $S_k = \left\{s^k \in \mathbb{R}^n \mid s^k_j = 0,\ j = 1,\ldots,k-1\right\}$, with root state and terminal state equal to $(0,\ldots,0)$
- Transition functions: $t_k(s^k,x_k) = (0,\ldots,0,s^{k+1}_{k+1},\ldots,s^{k+1}_n)$, where

$$s^{k+1}_\ell = \begin{Bmatrix} s^k_\ell + w_{k\ell}, \text{ if } x_k = \mathrm{S} \\ s^k_\ell - w_{k\ell}, \text{ if } x_k = \mathrm{T} \end{Bmatrix}, \quad \ell = k+1,\ldots,n$$

- Transition cost: $h_1(s^1,x_1) = 0$ for $x_1 \in \{\mathrm{S},\mathrm{T}\}$, and

$$h_k(s^k,x_k) = \begin{Bmatrix} (-s_k)^+ + \sum\limits_{\substack{\ell>k \\ s^j_\ell w_{j\ell} \le 0}} \min\left\{|s^j_\ell|, |w_{j\ell}|\right\}, \text{ if } x_k = \mathrm{S} \\ (s_k)^+ + \sum\limits_{\substack{\ell>k \\ s^j_\ell w_{j\ell} \ge 0}} \min\left\{|s^j_\ell|, |w_{j\ell}|\right\}, \quad \text{if } x_k = \mathrm{T} \end{Bmatrix}, \quad k = 2,\ldots,n$$

- Root value: $v_r = \sum\limits_{1\le j<j'\le n} (w_{jj'})^-$

Note that the root value is the sum of the negative arc weights. The state transition is based on the fact that, if vertex k is added to S, then the marginal benefit of placing vertex $\ell > k$ in T (given choices already made for vertices $1,\ldots,k-1$) is increased by $w_{k\ell}$. If k is added to T, the marginal benefit is reduced by $w_{k\ell}$. Figure 3.10 shows the resulting weighted BDD for the example discussed earlier.

Consider again the graph G in Fig. 3.9. Figure 3.10 depicts an exact BDD for the MCP on G with the node states as described before. A 0-arc leaving L_j indicates that $x_j = \mathrm{S}$, and a 1-arc indicates $x_j = \mathrm{T}$. Notice that the longest path p corresponds to the optimal solution $x^p = (\mathrm{S},\mathrm{S},\mathrm{T},\mathrm{S})$, and its length 4 is the weight of the maximum cut $(S,T) = (\{1,2,4\},\{3\})$.

3.10 Maximum 2-Satisfiability Problem

Let $x = (x_1,\ldots,x_n)$ be a tuple of Boolean variables, where each x_j can take value T or F (corresponding to true or false). A *literal* is a variable x_j or its negation $\neg x_j$. A *clause* c_i is a disjunction of literals, which is satisfied if at least one literal in c_i is true. If $C = \{c_1,\ldots,c_m\}$ is a set of clauses, each with exactly two literals, and if each c_i has weight $w_i \geq 0$, the *maximum 2-satisfiability problem* (MAX-2SAT) is the problem of finding an assignment of truth values to $x_1,\ldots,x_n$ that maximizes the sum of the weights of the satisfied clauses in C. MAX-2SAT has applications in scheduling, electronic design automation, computer architecture design, pattern recognition, inference in Bayesian networks, and many other areas [100, 105, 53].

To formulate the MAX-2SAT as a binary optimization problem, we use the Boolean variables x_j with domain $D_j = \{\mathrm{F},\mathrm{T}\}$. The constraint set $\mathscr{C}$ is empty, and

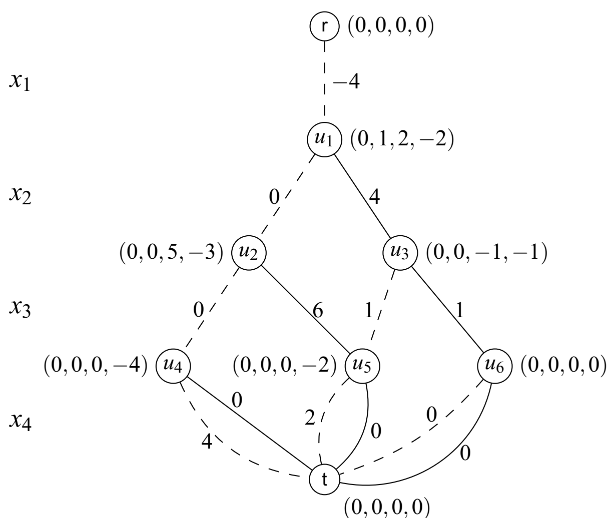


Fig. 3.10 Exact BDD with states for the MCP on the graph in Fig. 3.9.

Table 3.3 Data for a small MAX-2SAT problem.

Clause index	Clause	Weight
1	$x_1 \vee x_3$	3
2	$\neg x_1 \vee \neg x_3$	5
3	$\neg x_1 \vee x_3$	4
4	$x_2 \vee \neg x_3$	2
5	$\neg x_2 \vee \neg x_3$	1
6	$x_2 \vee x_3$	5

the objective function is $f(x) = \sum_{i=1}^{m} w_i c_i(x)$, where $c_i(x) = 1$ if x satisfies clause c_i, and $c_i(x) = 0$ otherwise. We thus write

$$\max \sum_{i=1}^{m} w_i c_i(x) \tag{3.13}$$
$$x_j \in \{\mathrm{F}, \mathrm{T}\}, \ \ j = 1, \dots, n.$$

Table 3.3 shows an example for an instance of MAX-2SAT with three Boolean variables x_1, x_2, and x_3. The optimal solution consists of setting $x = (\mathrm{F}, \mathrm{T}, \mathrm{T})$. It has length 19 since it satisfies all clauses but c_5.

To formulate MAX-2SAT as a DP model, we suppose without loss of generality that a MAX-2SAT problem contains all $4 \cdot \binom{n}{2}$ possible clauses, because missing clauses can be given zero weight. Thus $\mathscr{C}$ contains $x_j \vee x_k$, $x_j \vee \neg x_k$, $\neg x_j \vee x_k$, and $\neg x_j \vee \neg x_k$ for each pair $j, k \in \{1, \dots, n\}$ with $j \neq k$. Let w_{jk}^{TT} be the weight assigned to $x_j \vee x_k$, w_{jk}^{TF} the weight assigned to $x_j \vee \neg x_k$, and so forth.

We let each state variable s^k be an array $(s_1^k, \dots, s_n^k)$ in which each s_j^k is the net benefit of setting x_j to true, given previous settings. The net benefit is the advantage of setting $x_j = \mathrm{T}$ over setting $x_j = \mathrm{F}$. Suppose, for example, that $n = 2$ and we have fixed $x_1 = \mathrm{T}$. Then $x_1 \vee x_2$ and $x_1 \vee \neg x_2$ are already satisfied. The value of x_2 makes no difference for them, but setting $x_2 = \mathrm{T}$ newly satisfies $\neg x_1 \vee x_2$, while $x_2 = \mathrm{F}$ newly satisfies $\neg x_1 \vee \neg x_2$. Setting $x_2 = \mathrm{T}$ therefore obtains net benefit $w_{12}^{\mathrm{FT}} - w_{12}^{\mathrm{FF}}$. If x_1 has not yet been assigned a truth value, then we do not compute a net benefit for setting $x_2 = \mathrm{T}$. Formally, the DP formulation is as follows:

- State spaces: $S_k = \left\{ s^k \in \mathbb{R}^n \mid s_j^k = 0, \ j = 1, \dots, k-1 \right\}$, with root state and terminal state equal to $(0, \dots, 0)$
- Transition functions: $t_k(s^k, x_k) = (0, \dots, 0, s_{k+1}^{k+1}, \dots, s_n^{k+1})$, where

$$s_\ell^{k+1} = \left\{ \begin{array}{l} s_\ell^k + w_{k\ell}^{\mathrm{TT}} - w_{k\ell}^{\mathrm{TF}}, \text{ if } x_k = \mathrm{F} \\ s_\ell^k + w_{k\ell}^{\mathrm{FT}} - w_{k\ell}^{\mathrm{FF}}, \text{ if } x_k = \mathrm{T} \end{array} \right\}, \quad \ell = k+1, \ldots, n$$

- Transition cost: $h_1(s^1, x_1) = 0$ for $x_1 \in \{\mathrm{F}, \mathrm{T}\}$, and

$$h_k(s^k, x_k) = \left\{ \begin{array}{l} (-s_k^k)^+ + \sum_{\ell > k} \big(w_{k\ell}^{\mathrm{FF}} + w_{k\ell}^{\mathrm{FT}} + \\ \qquad \min\big\{ (s_\ell^k)^+ + w_{k\ell}^{\mathrm{TT}}, (-s_\ell^k)^+ + w_{k\ell}^{\mathrm{TF}} \big\} \big), \text{ if } x_k = \mathrm{F} \\ (s_k^k)^+ + \sum_{\ell > k} \big(w_{k\ell}^{\mathrm{TF}} + w_{k\ell}^{\mathrm{TT}} + \\ \qquad \min\big\{ (s_\ell^k)^+ + w_{k\ell}^{\mathrm{FT}}, (-s_\ell^k)^+ + w_{k\ell}^{\mathrm{FF}} \big\} \big), \text{ if } x_k = \mathrm{T} \end{array} \right\},$$

$$k = 2, \ldots, n$$

- Root value: $v_r = 0$

Figure 3.11 shows the resulting states and transition costs for the MAX-2SAT instance of Table 3.3. Notice that the longest path p yields the solution $x^p = (\mathrm{F}, \mathrm{T}, \mathrm{T})$ with length 14.

3.11 Compiling Decision Diagrams by Separation

Constraint separation is an alternative compilation procedure that modifies a DD iteratively until an exact representation is attained. It can be perceived as a method analogous to the separation procedures in integer programming (IP). In particular,

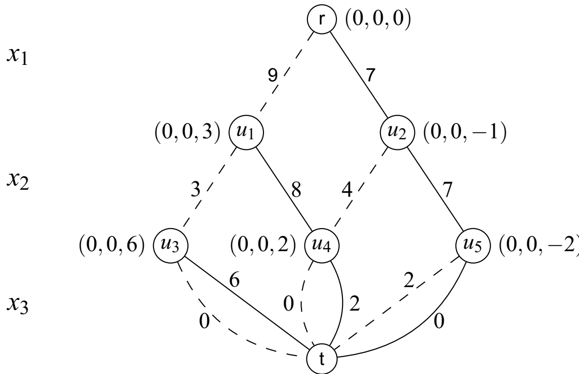


Fig. 3.11 Exact BDD with states for the MAX-2SAT problem of Table 3.3.

IP solvers typically enhance a continuous relaxation of the problem by adding *separating cuts* in the form of linear inequalities to the model if its optimal solution is infeasible. Such separating cuts may be general (such as *Gomory* cuts) or may exploit problem structure. In the same way, construction by separation considers separating cuts in a *discrete* relaxation of the problem. The separating cuts now take the form of *DP models* that are used to modify the DD instead of linear inequalities, and either can be general (e.g., separate arbitrary variable assignments) or may exploit problem structure.

Given a discrete optimization problem $\mathscr{P}$, the method starts with a DD B' that is a *relaxation* of the exact DD B: $\text{Sol}(B') \supseteq \text{Sol}(B) = \text{Sol}(\mathscr{P})$. That is, B' encodes all the feasible solutions to $\mathscr{P}$, but it may encode infeasible solutions as well. Each iteration consists of *separating a constraint* over B', i.e., changing the node and arc set of B' to remove the infeasible solutions that violate a particular constraint of the problem. The procedure ends when no more constraints are violated, or when the longest path according to given arc lengths is feasible to $\mathscr{P}$ (if one is only interested in the optimality). This separation method was first introduced in [84] for building approximate DDs in constraint programming models. The procedure in the original work, denoted by *incremental refinement*, is slightly different than that presented here and will be described in Section 4.7.

The outline of the method is depicted in Algorithm 2. The algorithm starts with a DD B' that is a relaxation of $\mathscr{P}$ and finds a constraint that is potentially violated by paths in B'. Such a constraint could be obtained by iterating on the original set of constraints of $\mathscr{P}$, or from some analysis of the paths of B'. The method now assumes that *each* constraint C is described by its own DP model having a state space and a transition function t_j^C. Since the states are particular to this constraint only, we associate a label $s(u)$ with each node u identifying the current state of u, which is reset to a value of χ every time a new constraint is considered. Hence, for notation purposes here, nodes are not identified directly with a state as in the top-down compilation method.

The next step of Algorithm 2 is to analyze each arc of B' separately in a top-down fashion. If the arc is infeasible according to t_j^C, it is removed from B' (nodes without incoming or outgoing arcs are assumed to be deleted automatically). If the endpoint of the arc is not associated with any state, it is then identified with the one that has the state given by t_j^C. Otherwise, if the endpoint of the arc is already associated with another state, we have to *split* the endpoint since each node in the DD must necessarily be associated with a single state. The splitting operation consists of

Algorithm 2 Exact DD Compilation by Separation

1: Let $B' = (U', A')$ be a DD such that $\mathrm{Sol}(B') \supseteq \mathrm{Sol}(\mathscr{P})$
2: **while** $\exists$ constraint C violated by B' **do**
3: Let $s(u) = \chi$ for all nodes $u \in B'$
4: $s(u) := \hat{r}$
5: **for** $j = 1$ to n **do**
6: **for** $u \in L_j$ **do**
7: **for** each arc $a = (u, v)$ leaving node u **do**
8: **if** $t_j^C(s(u), d(a)) \neq \hat{0}$ **then**
9: Remove arc a from B
10: **else if** $s(v) = \chi$ **then**
11: $s(v) = t_j^C(s(u), d(a))$
12: **else if** $s(v) \neq t_j^C(s(u), d(a))$ **then**
13: Remove arc (u, v)
14: Create new node v' with $s(v') = t_j^C(u, d(a))$
15: Add arc (u, v')
16: Copy outgoing arcs from v as outgoing arcs from v'
17: $L_j := L_j \cup \{v'\}$

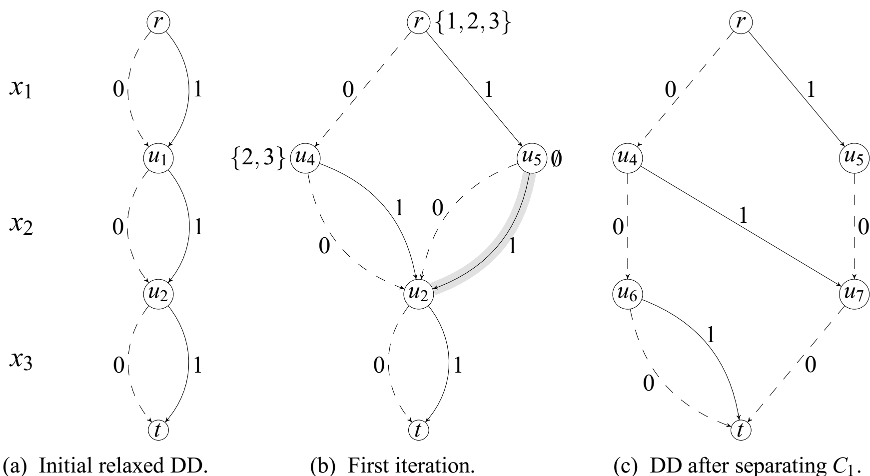
adding a new node to the layer, replicating the outgoing arcs from the original node (so that no solutions are lost), and resetting the endpoint of the arc to this new node. By performing these operations for all arcs of the DD, we ensure that constraint C is not violated by any solution encoded by B'. Transition costs could also be incorporated at any stage of the algorithm to represent an objective function of $\mathscr{P}$.

To illustrate the constraint separation procedure, consider the following optimization problem $\mathscr{P}$:

$$\begin{aligned} \max \quad & \sum_{i=1}^{3} x_i \\ & x_1 + x_2 \leq 1 \\ & x_2 + x_3 \leq 1 \\ & x_1 + x_3 \leq 1 \\ & x_1 + 2x_2 - 3x_3 \geq 2 \\ & x_1, x_2, x_3 \in \{0, 1\}. \end{aligned} \tag{3.14}$$

We partition the constraints into two sets:

$$C_1 = \{x_1 + x_2 \leq 1, x_2 + x_3 \leq 1, x_1 + x_3 \leq 1\} \text{ and } C_2 = \{x_1 + 2x_2 - 3x_3 \geq 2\}.$$



(a) Initial relaxed DD. (b) First iteration. (c) DD after separating C_1.

Fig. 3.12 First three iterations of the separation method for the problem (3.14).

We will compile the exact BDD for $\mathscr{P}$ by separating paths that violate constraint classes C_1 and C_2 in that order. The separation procedure requires a BDD encoding a relaxation of $\mathscr{P}$ as input. This can be trivially obtained by creating a 1-width BDD that contains the Cartesian product of the variable domains, as depicted in Fig. 3.12(a). The arc lengths have already been set to represent the transition costs.

We now separate the constraint set C_1. Notice that the inequalities in C_1 define the constraints of an independent set problem. Thus, we can directly use the state definition and transition function from Section 3.5 to separate C_1. Recall that the state in this case represents the variable indices that can still be added to the independent set so far. The state of the root node r is set to $s(r) = \{1,2,3\}$. We now process layer L_1. The 0-arc and the 1-arc leaving the root node lead to two distinct states $\{2,3\}$ and $\emptyset$, respectively. Hence, we split node u_1 into nodes u_4 and u_5 as depicted in Fig. 3.12(b), partitioning the incoming arcs and replicating the outgoing arcs so that no solutions are lost. Notice now that, according to the independent set transition function, the 1-arc leaving node u_5 leads to an infeasible state (shaded in Fig. 3.12(b)), therefore it will be removed when processing layer L_2.

The resulting DD after separating constraint C_1 is presented in Fig. 3.12(c). Notice that no solution violating C_1 is encoded in the DD. The separation procedure now repeats the same steps to separate constraint C_2, defining a suitable state and modifying the DD as necessary.

We remark that, in principle, any constraint C can be separated from a DD B' simply by *conjoining* B' with a second DD that represents all solutions satisfying C. DDs can be conjoined using a standard composition algorithm [157]. However, we have presented an algorithm that is designed specifically for separation and operates directly on the given DD. We do so for two reasons: (i) There is no need for an additional data structure to represent the second DD. (ii) The algorithm contains only logic that is essential to separation, which allows us to obtain a sharper bound on the size of the separating DD in structured cases.

There are some potential benefits of this procedure over the top-down approach of Section 3.4. First, it allows for an easier problem formulation since the combinatorial structure of each constraint can be considered separately when creating the DD, similarly to the modeling paradigm applied in constraint programming. Second, the separating constraints can be generated *dynamically*; for example, given arc lengths on a DD B', the constraints to be separated could be devised from an analysis of the longest path of B', perhaps considering alternative modeling approaches (such as logic-based Benders decomposition methods [96]). Finally, the DD B' that contains a feasible longest path could be potentially much smaller than the exact DD B for $\mathscr{P}$.

As in the top-down approach of Section 3.4, the resulting DD is not necessarily reduced. If this property is desired, one can reduce a DD by applying a single bottom-up procedure to identify equivalent nodes, as described in [157].

3.12 Correctness of the DP Formulations

In this section we show the correctness of the MCP and the MAX-2SAT formulations. The proof of correctness of the MISP formulation can be found in [23], the proof of correctness of the SCP and the SPP formulations in [27], and finally the proof of correctness of the MMP formulation in [49].

Theorem 3.1. *The specifications in Section 3.9 yield a valid DP formulation of the MCP.*

Proof. Note that any solution $x \in \{S,T\}^n$ is feasible, so that we need only show that condition (3.6) holds. The state transitions clearly imply that s^{n+1} is the terminal state $\hat{t} = (0,\ldots,0)$, and thus $s^{n+1} \in \{\hat{t},\hat{0}\}$. If we let $(\overline{s},\overline{x})$ be an arbitrary solution of (3.5), it remains to show that $\hat{f}(\overline{s},\overline{x}) = f(\overline{x})$. Let H_k be the sum of the first k

transition costs for solution $(\bar{s},\bar{x})$, so that $H_k = \sum_{j=1}^{k} h_j(\bar{s}^j,\bar{x}_j)$ and $H_n + v_r = \hat{f}(\bar{s},\bar{x})$. It suffices to show that

$$H_n + v_r = \sum_{j,j'} \left\{ w_{jj'} \mid 1 \le j < j' \le n,\, \bar{x}_j \ne \bar{x}_{j'} \right\}, \tag{3.15}$$

because the right-hand side is $f(\bar{x})$. We prove (3.15) as follows: Note first that the state transitions imply that

$$s_\ell^k = L_{k-1}^\ell - R_{k-1}^\ell, \ \text{ for } \ell \ge k, \tag{3.16}$$

where

$$L_k^\ell = \sum_{\substack{j \le k \\ \bar{x}_j = \mathrm{S}}} w_{j\ell}, \quad R_k^\ell = \sum_{\substack{j \le k \\ \bar{x}_j = \mathrm{T}}} w_{j\ell}, \ \text{ for } \ell > k.$$

We will show the following inductively:

$$H_k + N_k = \sum_{\substack{j < j' \le k \\ \bar{x}_j \ne \bar{x}_{j'}}} w_{jj'} + \sum_{\ell > k} \min\left\{ L_k^\ell, R_k^\ell \right\}, \tag{3.17}$$

where N_k is a partial sum of negative arc weights, specifically

$$N_k = \sum_{j < j' \le k} (w_{jj'})^- + \sum_{j \le k < \ell} (w_{k\ell})^-,$$

so that, in particular, $N_n = v_r$. This proves the theorem, because (3.17) implies (3.15) when $k = n$.

We first note that (3.17) holds for $k = 1$, because in this case both sides vanish. We now suppose (3.17) holds for $k-1$ and show that it holds for k. The definition of transition cost implies

$$H_k = H_{k-1} + (\sigma_k s_k^k)^+ + \sum_{\substack{\ell > k \\ \sigma_k s_\ell^k w_{k\ell} \ge 0}} \min\left\{ |s_\ell^k|, |w_{k\ell}| \right\},$$

where σ_k is 1 if $\bar{x}_k = \mathrm{T}$ and -1 otherwise. This and the inductive hypothesis imply

$$H_k = \sum_{\substack{j < j' \le k-1 \\ \bar{x}_j \ne \bar{x}_{j'}}} w_{jj'} + \sum_{\ell \ge k} \min\left\{ L_{k-1}^\ell, R_{k-1}^\ell \right\} - N_{k-1} + (\sigma_k s_k^k)^+ + \sum_{\substack{\ell > k \\ \sigma_k s_\ell^k w_{k\ell} \ge 0}} \min\left\{ |s_\ell^k|, |w_{k\ell}| \right\}.$$

We wish to show that this is equal to the right-hand side of (3.17) minus N_k. Making the substitution (3.16) for state variables, we can establish this equality by showing

$$\sum_{\ell \geq k} \min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell}\right\} - N_{k-1} + (\sigma_k(L_{k-1}^{k} - R_{k-1}^{k}))^{+} + \sum_{\substack{\ell > k \\ \sigma_k(L_{k-1}^{\ell} - R_{k-1}^{\ell})w_{k\ell} \geq 0}} \min\left\{|L_{k-1}^{\ell} - R_{k-1}^{\ell}|, |w_{k\ell}|\right\}$$
$$= \sum_{\substack{j<k \\ \bar{x}_j \neq \bar{x}_k}} w_{jk} + \sum_{\ell > k} \min\left\{L_k^{\ell}, R_k^{\ell}\right\} - N_k. \tag{3.18}$$

We will show that (3.18) holds when $\bar{x}_k = \mathrm{T}$. The proof for $\bar{x}_k = \mathrm{S}$ is analogous. Using the fact that $R_k^{\ell} = R_{k-1}^{\ell} + w_{k\ell}$, (3.18) can be written

$$\min\left\{L_{k-1}^{k}, R_{k-1}^{k}\right\} + \sum_{\ell > k} \min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell}\right\} + (L_{k-1}^{k} - R_{k-1}^{k})^{+}$$
$$+ \sum_{\substack{\ell > k \\ (L_{k-1}^{\ell} - R_{k-1}^{\ell})w_{k\ell} \geq 0}} \min\left\{|L_{k-1}^{\ell} - R_{k-1}^{\ell}|, |w_{k\ell}|\right\} \tag{3.19}$$
$$= L_{k-1}^{k} + \sum_{\ell > k} \min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell} + w_{k\ell}\right\} - (N_k - N_{k-1}).$$

The first and third terms of the left-hand side of (3.19) sum to L_{k-1}^{k}. We can therefore establish (3.19) by showing that, for each $\ell \in \{k+1, \ldots, n\}$, we have

$$\min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell}\right\} + \delta \min\left\{R_{k-1}^{\ell} - L_{k-1}^{\ell}, -w_{k\ell}\right\}$$
$$= \min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell} + w_{k\ell}\right\} - w_{k\ell}, \quad \text{if } w_{k\ell} < 0$$
$$\min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell}\right\} + (1-\delta) \min\left\{L_{k-1}^{\ell} - R_{k-1}^{\ell}, w_{k\ell}\right\}$$
$$= \min\left\{L_{k-1}^{\ell}, R_{k-1}^{\ell} + w_{k\ell}\right\}, \quad \text{if } w_{k\ell} \geq 0$$

where $\delta = 1$ if $L_{k-1}^{\ell} \leq R_{k-1}^{\ell}$ and $\delta = 0$ otherwise. It is easily checked that both equations are identities.

Theorem 3.2. *The specifications in Section 3.10 yield a valid DP formulation of the MAX-2SAT problem.*

Proof. Since any solution $x \in \{F, T\}^n$ is feasible, we need only show that the costs are correctly computed. Thus if $(\bar{s}, \bar{x})$ is an arbitrary solution of (3.5), we wish to show that $\hat{f}(\bar{s}, \bar{x}) = f(\bar{x})$. If H_k is as before, we wish to show that $H_n = \mathrm{SAT}_n(\bar{x})$, where $\mathrm{SAT}_k(\bar{x})$ is the total weight of clauses satisfied by the settings $\bar{x}_1, \ldots, \bar{x}_k$. Thus

$$\text{SAT}_k(\bar{x}) = \sum_{jj'\alpha\beta} \{w_{jj'}^{\alpha\beta} \mid 1 \le j < j' \le k;\ \alpha,\beta \in \{\text{F},\text{T}\};\ \bar{x}_j = \alpha \text{ or } \bar{x}_{j'} = \beta\}.$$

Note first that the state transitions imply (3.16) as in the previous proof, where

$$L_k^\ell = \sum_{\substack{1\le j\le k \\ \bar{x}_j=\text{T}}} w_{j\ell}^{\text{FT}} + \sum_{\substack{1\le j\le k \\ \bar{x}_j=\text{F}}} w_{j\ell}^{\text{TT}}, \quad R_k^\ell = \sum_{\substack{1\le j\le k \\ \bar{x}_j=\text{T}}} w_{j\ell}^{\text{FF}} + \sum_{\substack{1\le j\le k \\ \bar{x}_j=\text{F}}} w_{j\ell}^{\text{TF}}, \text{ for } \ell > k,$$

We will show the following inductively:

$$H_k = \text{SAT}_k(\bar{x}) + \sum_{\ell>k} \min\left\{L_k^\ell, R_k^\ell\right\}, \tag{3.20}$$

This proves the theorem, because (3.20) reduces to $H_n = \text{SAT}_n(\bar{x})$ when $k = n$.

To simplify the argument, we begin the induction with $k = 0$, for which both sides of (3.20) vanish. We now suppose (3.20) holds for $k-1$ and show that it holds for k. The definition of transition cost implies

$$H_k = H_{k-1} + (\sigma_k s_k^k)^+ + \sum_{\ell>k}\left(w_{k\ell}^{\alpha\text{F}} + w_{k\ell}^{\alpha\text{T}} + \min\left\{(s_\ell^k)^+ + w_{k\ell}^{\beta\text{T}}, (-s_\ell^k)^+ + w_{k\ell}^{\beta\text{F}}\right\}\right),$$

where σ_k is 1 if $\bar{x}_k = \text{T}$ and -1 otherwise. Also α is the truth value $\bar{x}_k$ and β is the value opposite $\bar{x}_k$. This and the inductive hypothesis imply

$$\begin{aligned} H_k = \text{SAT}_{k-1}(\bar{x}) &+ \sum_{\ell\ge k} \min\left\{L_k^\ell, R_k^\ell\right\} + (\sigma_k s_k^k)^+ \\ &+ \sum_{\ell>k}\left(w_{k\ell}^{\alpha\text{F}} + w_{k\ell}^{\alpha\text{T}} + \min\left\{(s_\ell^k)^+ + w_{k\ell}^{\beta\text{T}}, (-s_\ell^k)^+ + w_{k\ell}^{\beta\text{F}}\right\}\right). \end{aligned}$$

We wish to show that this is equal to the right-hand side of (3.20). We will establish this equality on the assumption that $\bar{x}_k = \text{T}$, as the proof is analogous when $\bar{x}_k = \text{F}$. Making the substitution (3.16) for state variables, and using the facts that $L_k^\ell = L_{k-1}^\ell + w_{k\ell}^{\text{FT}}$ and $R_k^\ell = R_{k-1}^\ell + w_{k\ell}^{\text{FF}}$, it suffices to show

$$\begin{aligned} &\sum_{\ell>k} \min\left\{L_k^\ell, R_k^\ell\right\} + \min\left\{L_{k-1}^k, R_{k-1}^k\right\} + (L_{k-1}^k - R_{k-1}^k)^+ \\ &\quad + \sum_{\ell>k}\left(w_{k\ell}^{\text{TF}} + w_{k\ell}^{\text{TT}} + \min\left\{(L_{k-1}^\ell - R_{k-l}^\ell)^+ + w_{k\ell}^{\text{FT}}, (L_{k-1}^\ell - R_{k-l}^\ell)^+ + w_{k\ell}^{\text{FF}}\right\}\right) \\ &= \sum_{\ell>k} \min\left\{L_{k-1}^\ell + w_{k\ell}^{\text{FT}}, R_k^\ell + w_{k\ell}^{\text{FF}}\right\} + \text{SAT}_k(\bar{x}) - \text{SAT}_{k-1}(\bar{x}). \end{aligned} \tag{3.21}$$

The second and third terms of the left-hand side of (3.21) sum to L^k_{k-1}. Also

$$\mathrm{SAT}_k(\overline{x}) - \mathrm{SAT}_{k-1}(\overline{x}) = L^k_{k-1} + \sum_{\ell>k} \left(w^{\mathrm{TF}}_{k\ell} + w^{\mathrm{TT}}_{k\ell} \right).$$

We can therefore establish (3.21) by showing that

$$\begin{aligned}\min\left\{L^\ell_{k-1}, R^\ell_{k-1}\right\} + \min\left\{(L^\ell_{k-1} - R^\ell_{k-1})^+ + w^{\mathrm{FT}}_{k\ell}, (R^\ell_{k-1} - L^\ell_{k-1})^+ + w^{\mathrm{FF}}_{k\ell}\right\} \\ = \min\left\{L^\ell_{k-1} + w^{\mathrm{FT}}_{k\ell}, R^\ell_{k-1} + w^{\mathrm{FF}}_{k\ell}\right\}\end{aligned}$$

for $\ell > k$. It can be checked that this is an identity.

Chapter 4
Relaxed Decision Diagrams

Abstract Bounds on the optimal value are often indispensable for the practical solution of discrete optimization problems, as for example in branch-and-bound procedures. This chapter explores an alternative strategy of obtaining bounds through *relaxed decision diagrams*, which overapproximate both the feasible set and the objective function of the problem. We first show how to modify the top-down compilation from the previous chapter to generate relaxed decision diagrams. Next, we present three modeling examples for classical combinatorial optimization problems, and provide a thorough computational analysis of relaxed diagrams for the maximum independent set problem. The chapter concludes by describing an alternative method to generate relaxed diagrams, the *incremental refinement procedure*, and exemplify its application to a single-machine makespan problem.

4.1 Introduction

A weighted DD B is *relaxed* for an optimization problem $\mathscr{P}$ if B represents a superset of the feasible solutions of $\mathscr{P}$, and path lengths are upper bounds on the value of feasible solutions. That is, B is relaxed for $\mathscr{P}$ if

$$\mathrm{Sol}(\mathscr{P}) \subseteq \mathrm{Sol}(B), \tag{4.1}$$

$$f(x^p) \leq v(p), \text{ for all } r\text{–}t \text{ paths } p \text{ in } B \text{ for which } x^p \in \mathrm{Sol}(\mathscr{P}). \tag{4.2}$$

Suppose $\mathscr{P}$ is a maximization problem. In Chapter 3, we showed that an exact DD reduces discrete optimization to a longest-path problem: If p is a longest path in

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_4

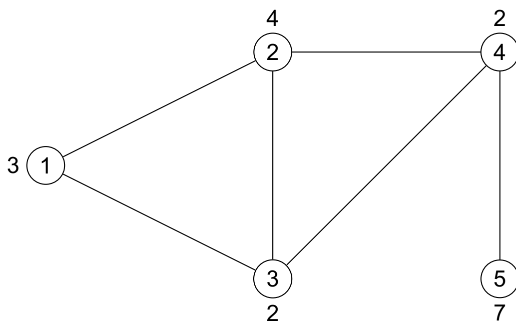


Fig. 4.1 Graph with vertex weights for the MISP.

a BDD B that is exact for $\mathscr{P}$, then x^p is an optimal solution of $\mathscr{P}$, and its length $v(p)$ is the optimal value $z^*(\mathscr{P}) = f(x^p)$ of $\mathscr{P}$. When B is relaxed for $\mathscr{P}$, a longest path p provides an *upper bound* on the optimal value. The corresponding solution x^p may not be feasible, but $v(p) \geq z^*(\mathscr{P})$. We will show that the width of a relaxed DD is restricted by an input parameter, which can be adjusted according to the number of variables of the problem and computer resources.

Consider the graph and vertex weights depicted in Fig. 4.1. Figure 4.2(a) represents an exact BDD in which each path corresponds to an independent set encoded by the arc labels along the path, and each independent set corresponds to some path.

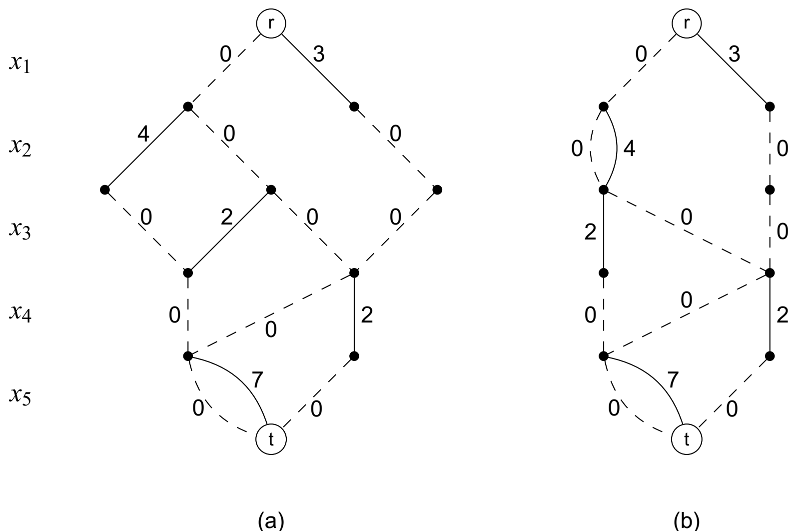


Fig. 4.2 (a) Exact BDD and (b) relaxed BDD for the MISP on the graph in Fig. 4.1.

A 1-arc leaving layer L_j (solid) indicates that vertex j is in the independent set, and a 0-arc (dashed) indicates that it is not. The longest *r*–*t* path in the BDD has value 11, corresponding to solution $x = (0,1,0,0,1)$ and to the independent set $\{2,5\}$, the maximum-weight independent set in the graph.

Figure 4.2(b) shows a relaxed BDD. Each independent set corresponds to a path, but there are paths p for which x^p is infeasible (i.e., not an independent set). For example, the path $\bar{p}$ encoding $x^{\bar{p}} = (0,1,1,0,1)$ does not represent an independent set because both endpoints of edge $(2,3)$ are selected. The length of each path that represents an independent set is the weight of that set, making this a relaxed BDD. The longest path in the BDD is $\bar{p}$, providing an upper bound of 13.

Relaxed DDs were introduced by [4] for the purpose of replacing the domain store used in constraint programming by a richer data structure. Similar methods were applied to other types of constraints [84, 85, 94], all of which apply an alternative method of generating relaxation denoted by *incremental refinement*, described in Chapter 9. In this chapter we derive relaxed DDs directly from a DP formulation of the problem. Weighted DD relaxations were used to obtain optimization bounds in [28, 24], the former of which applied them to set covering and the latter to the maximum independent set problem.

4.2 Top-Down Compilation of Relaxed DDs

Relaxed DDs of limited width can be built by considering an additional step in the modeling framework for *exact DDs* described in Section 3.4. Recall that such a framework relies on a DP model composed of a state space, transition functions, transition cost functions, and a root value. For relaxed DDs, the model should also have an additional rule describing how to *merge* nodes in a layer to ensure that the output DD will satisfy conditions (4.1) and (4.2), perhaps with an adjustment in the transition costs. The underlying goal of this rule is to create a relaxed DD that provides a tight bound given the maximum available width.

This rule is applied in the following way. When a layer L_j in the DD grows too large during a top-down construction procedure, we heuristically select a subset $M \subseteq L_j$ of nodes in the layer to be merged, perhaps by choosing nodes with similar states. The state of the merged nodes is defined by an operator $\oplus(M)$, and the length v of every arc coming into a node $u \in M$ is modified to $\Gamma_M(v,u)$. The process is repeated until $|L_j|$ no longer exceeds the maximum width W.

Algorithm 3 Relaxed DD Top-Down Compilation for Maximum Width W

1: Create node $r = \hat{r}$ and let $L_1 = \{r\}$
2: **for** $j = 1$ to n **do**
3: **while** $|L_j| > W$ **do**
4: let $M = \text{node_select}(L_j)$, $L_j \leftarrow (L_j \setminus M) \cup \{\oplus(M)\}$
5: **for all** $u \in L_{j-1}$ and $i \in D_j$ with $b_i(u) \in M$ **do**
6: $b_i(u) \leftarrow \oplus(M)$, $v(a_i(u)) \leftarrow \Gamma_M(v(a_i(u)), b_i(u))$
7: let $L_{j+1} = \emptyset$
8: **for all** $u \in L_j$ and $d \in D(x_j)$ **do**
9: **if** $t_j(u,d) \neq \hat{0}$ **then**
10: let $u' = t_j(u,d)$, add u' to L_{j+1}, and set $b_d(u) = u'$, $v(u,u') = h_j(u,u')$
11: Merge nodes in L_{n+1} into terminal node t

The relaxed DD construction procedure is formally presented in Algorithm 3. The algorithm uses the notation $a_v(u)$ as the arc leaving node u with label v, and $b_v(u)$ to denote the node at the opposite end of the arc leaving node u with value v (if it exists). The algorithm identifies a node u with the DP state associated with it. The relaxed DD construction is similar to the exact DD construction procedure depicted in Algorithm 1, except for the addition of lines 3 to 6 to account for the node merging rule. Namely, if the layer L_j size exceeds the maximum allotted width W, a heuristic function *node_select* selects a subset of nodes M. The nodes in M are merged into a new node with state $\oplus(M)$. The incoming arcs in M are redirected to $\oplus(M)$, and their transition costs are modified according to $\Gamma_M(v,u)$. This procedure is repeated until $|L_j| \leq W$; when that is the case, the algorithm then follows the same steps as in the exact DD construction of Algorithm 1.

The two key operations in a relaxed DD construction are thus the merge operator $\oplus(M)$ and the node selection rule (represented by the function *node_select* in Algorithm 3). While the first must ensure that the relaxed DD is indeed a valid relaxation according to conditions (4.1) and (4.2), the second directly affects the quality of the optimization bounds provided by relaxed DDs. We will now present valid relaxation operators $\oplus(M)$ for the maximum independent set, the maximum cut problem, and the maximum 2-satisfiability problem. In Section 4.6 we present a computational analysis of how the choice of nodes to merge influences the resulting optimization bound for the maximum independent set problem.

4.3 Maximum Independent Set

The maximum independent set problem (MISP), first presented in Section 3.5, can be summarized as follows: Given a graph $G = (V,E)$ with an arbitrarily ordered vertex set $V = \{1,2,\ldots,n\}$ and weight $w_j \geq 0$ for each vertex $j \in V$, we wish to find a maximum-weight set $I \subseteq V$ such that no two vertices in I are connected by an edge in E. It is formulated as the following discrete optimization problem:

$$
\begin{aligned}
\max \ & \sum_{j=1}^{n} w_j x_j \\
& x_i + x_j \leq 1, \ \text{for all } (i,j) \in E \\
& x_j \in \{0,1\}, \ \text{for all } j \in V.
\end{aligned}
$$

In the DP model for the MISP, recall from Section 3.5 that the state associated with a node is the set of vertices that can still be added to the independent set. That is,

- State spaces: $S_j = 2^{V_j}$ for $j = 2,\ldots,n$, $\hat{r} = V$, and $\hat{t} = \emptyset$
- Transition functions: $t_j(s^j,0) = s^j \setminus \{j\}$, $t_j(s^j,1) = \begin{cases} s^j \setminus N(j), & \text{if } j \in s^j \\ \hat{0}, & \text{if } j \notin s^j \end{cases}$
- Cost functions: $h_j(s^j,0) = 0$, $h_j(s^j,1) = w_j$
- A root value of 0

To create a relaxed DD for the MISP, we introduce a merging rule into the DP model above where states are merged simply by taking their union. Hence, if $M = \{u_i \mid i \in I\}$, the merged state is $\oplus(M) = \bigcup_{i \in I} u_i$. The transition cost is not changed, so that $\Gamma_M(v,u) = v$ for all v,u. The correctness follows from the fact that a transition function leads to an infeasible state only if a vertex j is not in s^j, therefore no solutions are lost.

As an example, Fig. 4.3(a) presents an exact DD for the MISP instance defined on the graph in Fig. 4.1. Figure 4.3(b) depicts a relaxed BDD for the same graph with a maximum width of 2, where nodes u_2 and u_3 are merged during the top-down procedure to obtain $u' = u_2 \cup u_3 = \{3,4,5\}$. This reduces the BDD width to 2.

In the exact DD of Fig. 4.3(a), the longest path p corresponds to the optimal solution $x^p = (0,1,0,0,1)$ and its length 11 is the weight of the maximum independent set $\{2,5\}$. In the relaxed DD of Fig. 4.3(b), the longest path corresponds to the solution $(0,1,1,0,1)$ and has length 13, which provides an upper bound of 13 on the

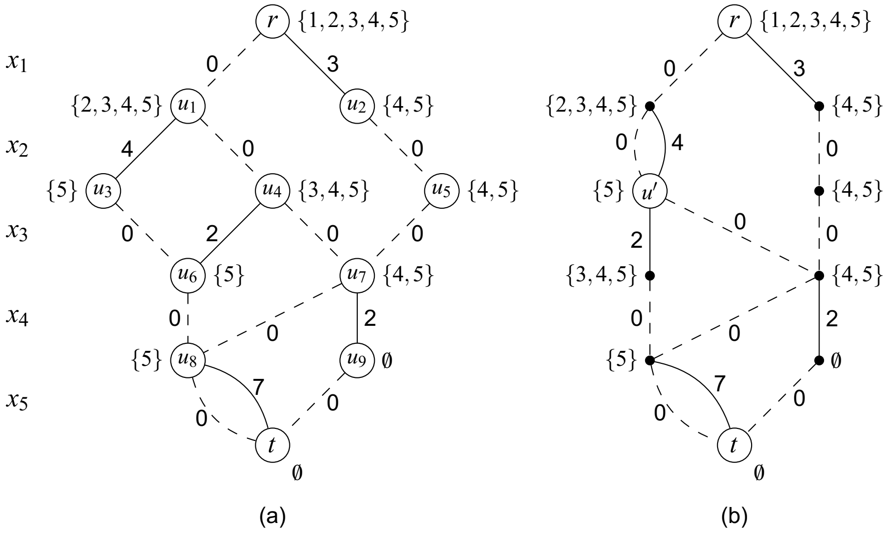


Fig. 4.3 (a) Exact BDD with states for the MISP on the graph in Fig. 4.1. (b) Relaxed BDD for the same problem instance.

objective function. Note that the longest path in the relaxed DD corresponds to an infeasible solution to that instance.

4.4 Maximum Cut Problem

The maximum cut problem (MCP) was first presented in Section 3.9. Given a graph $G = (V,E)$ with vertex set $V = \{1,\dots,n\}$, a cut (S,T) is a partition of the vertices in V. We say that an edge crosses the cut if its endpoints are on opposite sides of the cut. Given edge weights, the value $v(S,T)$ of a cut is the sum of the weights of the edges crossing the cut. The MCP is the problem of finding a cut of maximum value.

The DP model for the MCP in Section 3.9 considers a state that represents the net benefit of adding vertex ℓ to set T. That is, using the notation $(\alpha)^+ = \max\{\alpha,0\}$ and $(\alpha)^- = \min\{\alpha,0\}$, the DP model was:

- State spaces: $S_k = \left\{s^k \in \mathbb{R}^n \mid s^k_j = 0,\ j = 1,\dots,k-1\right\}$, with root state and terminal state equal to $(0,\dots,0)$
- Transition functions: $t_k(s^k,x_k) = (0,\dots,0,s^{k+1}_{k+1},\dots,s^{k+1}_n)$, where

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

- Transition cost: $h_1(s^1, x_1) = 0$ for $x_1 \in \{\text{S}, \text{T}\}$, and

$$h_k(s^k, x_k) = \left\{ \begin{array}{ll} (-s_k)^+ + \sum\limits_{\substack{\ell > k \\ s_\ell^j w_{j\ell} \le 0}} \min\left\{|s_\ell^j|, |w_{j\ell}|\right\}, & \text{if } x_k = \text{S} \\ (s_k)^+ + \sum\limits_{\substack{\ell > k \\ s_\ell^j w_{j\ell} \ge 0}} \min\left\{|s_\ell^j|, |w_{j\ell}|\right\}, & \text{if } x_k = \text{T} \end{array} \right\}, \quad k = 2, \ldots, n$$

- Root value: $v_r = \sum\limits_{1 \le j < j' \le n} (w_{jj'})^-$

Recall that we identify each node $u \in L_k$ with the associated state vector s^k. When we merge two nodes u^1 and u^2 in a layer L_k, we would like the resulting node $u^{\text{new}} = \oplus(\{u, u'\})$ to reflect the values in u and u' as closely as possible, while resulting in a valid relaxation. In particular, path lengths should not decrease. Intuitively, it may seem that $u_j^{\text{new}} = \max\{u_j^1, u_j^2\}$ for each j is a valid relaxation operator, because increasing state values could only increase path lengths. However, this can reduce path lengths as well. It turns out that we can offset any reduction in path lengths by adding the absolute value of the state change to the length of incoming arcs.

This yields the following procedure for merging nodes in M. If, for a given ℓ, the states u_ℓ have the same sign for all nodes $u \in M$, we change each u_ℓ to the state with smallest absolute value, and add the absolute value of each change to the length of arcs entering u. When the states u_ℓ differ in sign, we change each u_ℓ to zero and again add the absolute value of the changes to incoming arcs. More precisely, when $M \subset L_k$ we let

$$\oplus(M)_\ell = \left\{ \begin{array}{ll} \min\limits_{u \in M} \{u_\ell\}, & \text{if } u_\ell \ge 0 \text{ for all } u \in M \\ -\min\limits_{u \in M} \{|u_\ell|\}, & \text{if } u_\ell \le 0 \text{ for all } u \in M \\ 0, & \text{otherwise} \end{array} \right\}, \ \ell = k, \ldots, n \qquad \text{(MCP-relax)}$$

$$\Gamma_M(v, u) = v + \sum_{\ell \ge k} \left(|u_\ell| - |\oplus(M)_\ell|\right), \ \text{ all } u \in M.$$

Figure 4.5 shows an exact DD and a relaxed DD for the MCP instance defined over the graph in Fig. 4.4. The relaxed DD is obtained by merging nodes u_2 and

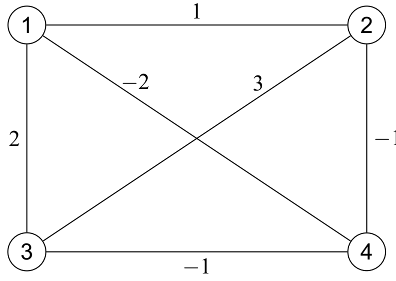


Fig. 4.4 Graph with edge weights for the MCP.

u_3 during top-down construction. In the exact DD of Fig. 4.5(a), the longest path p corresponds to the optimal solution $x^p = (\mathrm{S}, \mathrm{S}, \mathrm{T}, \mathrm{S})$, and its length 4 is the weight of the maximum cut $(S,T) = (\{1,2,4\},\{3\})$. In the relaxed DD of Fig. 4.5(b), the longest path corresponds to the solution $(\mathrm{S}, \mathrm{S}, \mathrm{S}, \mathrm{S})$ and provides an upper bound of 5 on the objective function. Note that the actual weight of this cut is 0.

To show that $\oplus$ and Γ are valid relaxation operators, we rely on Lemma 4.1:

Lemma 4.1. *Let B be an exact BDD generated by Algorithm 1 for an instance $\mathscr{P}$ of the MCP. Suppose we add Δ to one state s_ℓ^k in layer k of B ($\ell \geq k$), and add $|\Delta|$ to the length of each arc entering the node u associated with s^k. If we then recompute layers $k,\ldots,n+1$ of B as in Algorithm 3, the result is a relaxed BDD for $\mathscr{P}$.*

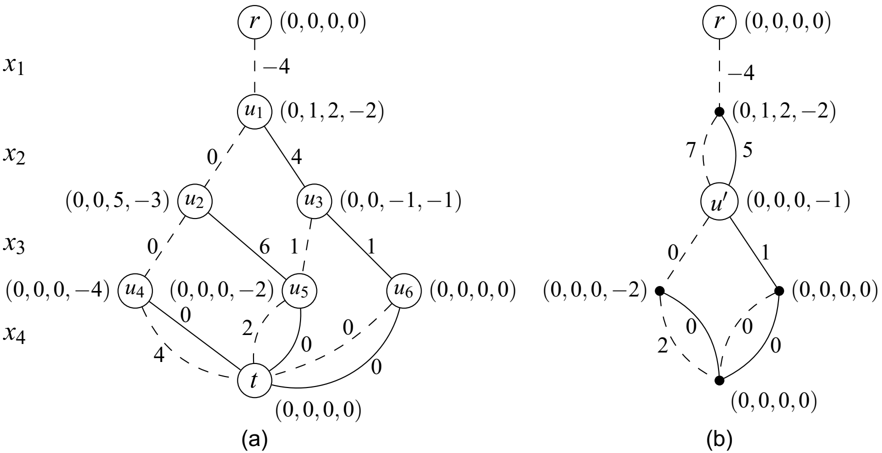


Fig. 4.5 (a) Exact BDD with states for the MCP on the graph in Fig. 4.4. (b) Relaxed BDD for the same problem instance.

Proof. Let B' be the result of recomputing the BDD, and take any $\bar{x} \in \{\mathrm{S},\mathrm{T}\}^n$. It suffices to show that the path p corresponding to $\bar{x}$ is no shorter in B' than in B. We may suppose p contains u, because otherwise p has the same length in B and B'. Only arcs of p that leave layers $L_{k-1},\ldots,L_n$ can have different lengths in B'. The length $v(a)$ of the arc a leaving L_{k-1} becomes $v(a)+|\Delta|$. The states s_ℓ^j along p in B for $j=k,\ldots,n$ become $s_\ell^j+\Delta$ in B', and all other states along p are unchanged. Thus from the formula for transition cost, the length $v(a')$ of the arc a' leaving L_ℓ becomes at least

$$
\begin{aligned}
&v(a')+\min\left\{(-(s_\ell^\ell+\Delta))^+,(s_\ell^\ell+\Delta)^+\right\}-\min\left\{(-s_\ell^\ell)^+,(s_\ell^\ell)^+\right\}\\
&\qquad\geq v(a')+\min\left\{(-s_\ell^\ell)^+-\Delta,(s_\ell^\ell)^+ +\Delta\right\}-\min\left\{(-s_\ell^\ell)^+,(s_\ell^\ell)^+\right\}\\
&\qquad\geq v(a')-|\Delta|.
\end{aligned}
$$

From the same formula, the lengths of arcs leaving L_j for $j>k$ and $j\neq\ell$ cannot decrease. As a result, the length $v(p)$ of p in B becomes at least $v(p)+|\Delta|-|\Delta| = v(p)$ in B'. □

Theorem 4.1. *Operators $\oplus$ and Γ as defined in (MCP-relax) are valid relaxation operators for the MCP.*

Proof. We can achieve the effect of Algorithm 3 if we begin with the exact BDD, successively alter only one state s_ℓ^k and the associated incoming arc lengths as prescribed by (MCP-relax), and compute the resulting exact BDD after each alteration. We begin with states in L_2 and work down to L_n. In each step of this procedure, we increase or decrease $s_\ell^k = u_\ell$ by $\delta = |u_\ell| - |\oplus(M)_\ell|$ for some $M \subset L_k$, where $\oplus(M)_\ell$ is computed using the states that were in L_k immediately after all the states in L_{k-1} were updated. We also increase the length of arcs into u_ℓ by δ. This means we can let $\Delta = \pm\delta$ in Lemma 4.1 and conclude that each step of the procedure yields a relaxed BDD. □

4.5 Maximum 2-Satisfiability Problem

The maximum 2-satisfiability problem was described in Section 3.10. The interpretation of states is very similar for the MCP and MAX-2SAT. We therefore use the

same relaxation operators (MCP-relax). The proof of their validity for MAX-2SAT is analogous to the proof of Theorem 4.1.

Theorem 4.2. *Operators $\oplus$ and Γ as defined in (MCP-relax) are valid relaxation operators for MAX-2SAT.*

4.6 Computational Study

In this section we assess empirically the quality of bounds provided by a relaxed BDD. We first investigate the impact of various parameters on the bounds. We then compare our bounds with those obtained by a linear programming (LP) relaxation of a clique-cover model of the problem, both with and without cutting planes. We measure the quality of a bound by its ratio with the optimal value (or best lower bound known if the problem instance is unsolved). Thus a smaller ratio indicates a better bound.

We test our procedure on two sets of instances. The first set, denoted by `random`, consists of 180 randomly generated graphs according to the Erdös–Rényi model $G(n,p)$, in which each pair of n vertices is joined by an edge with probability p. We fix $n = 200$ and generate 20 instances for each $p \in \{0.1, 0.2, \ldots, 0.9\}$. The second set of instances, denoted by `dimacs`, is composed by the complement graphs of the well-known DIMACS benchmark for the maximum clique problem, obtained from `http://cs.hbg.psu.edu/txn131/clique.html`. These graphs have between 100 and 4000 vertices and exhibit various types of structure. Furthermore, we consider the maximum cardinality optimization problem for our test bed (i.e., $w_j = 1$ for all vertices v_j).

The tests ran on an Intel Xeon E5345 with 8 GB RAM in single-core mode. The BDD method was implemented in C++.

4.6.1 Merging Heuristics

The selection of nodes to merge in a layer that exceeds the maximum allotted width W is critical for the construction of relaxed BDDs. Different selections may yield dramatic differences in the obtained upper bounds on the optimal value, since the merging procedure adds paths corresponding to infeasible solutions to the BDD.

We now present a number of possible heuristics for selecting nodes. This refers to how the subsets M are chosen according to the function *node_select* in Algorithm 3. The heuristics we test are described below.

- **random**: Randomly select a subset M of size $|L_j| - W + 1$ from L_j. This may be used as a stand-alone heuristic or combined with any of the following heuristics for the purpose of generating several relaxations.
- **minLP**: Sort nodes in L_j in increasing order of the longest path value up to those nodes and merge the first $|L_j| - W + 1$ nodes. This is based on the idea that infeasibility is introduced into the BDD only when nodes are merged. By selecting nodes with the smallest longest path, we lose information in parts of the BDD that are unlikely to participate in the optimal solution.
- **minSize**: Sort nodes in L_j in decreasing order of their corresponding state sizes and merge the first two nodes until $|L_j| \leq W$. This heuristic merges nodes that have the largest number of vertices in their associated states. Because larger vertex sets are likely to have more vertices in common, the heuristic tends to merge nodes that represent similar regions of the solution space.

We evaluated these three merging heuristics on the `random` instance benchmark by considering a maximum width of $W = 10$ and using the same BDD variable ordering heuristic for all cases. Specifically, we applied a *Maximal Path Decomposition* variable ordering (`MPD`), to be described in details in Section 7.3. Figure 4.6 displays the resulting bound quality.

We see that, among the merging heuristics tested, **minLP** achieves by far the tightest bounds. This behavior reflects the fact that infeasibility is introduced only at those nodes selected to be merged, and it seems better to preserve the nodes with the best bounds as in **minLP**. The plot also highlights the importance of using a structured merging heuristic, since **random** yielded much weaker bounds than the other techniques tested. In light of these results, we used **minLP** as the merging heuristic for the remainder of the experiments.

4.6.2 Variable Ordering Heuristic

As will be studied in detail in Chapter 7, the ordering of the vertices plays an important role in not only the size of exact BDDs, but also in the bound obtained by relaxed BDDs.

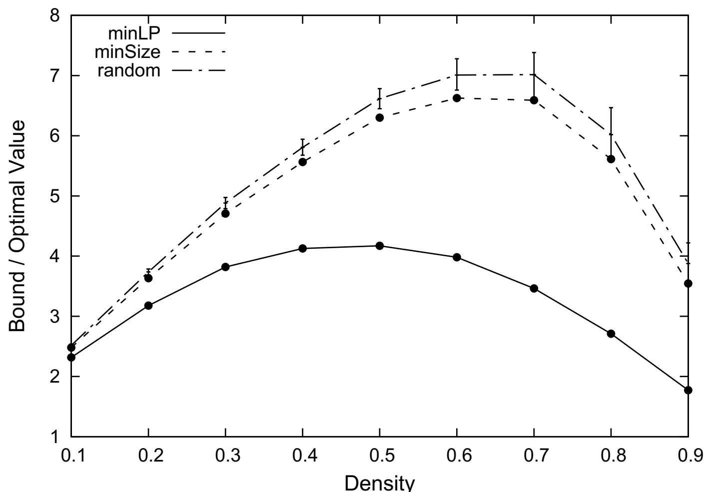


Fig. 4.6 Bound quality vs. graph density for each merging heuristic, using the `random` instance set with `MPD` ordering and maximum BDD width 10. Each data point represents an average over 20 problem instances. The vertical line segments indicate the range obtained in five trials of the **random** heuristic.

For our experiments we apply a dynamic ordering denoted by **minState**. Supposing that we have already built layers $L_1, \ldots, L_{j-1}$, we select the vertex v_j appearing in the fewest number of states associated with the nodes of the last layer L_{j-1}. This intuitively minimizes the size of L_j, since the nodes that must be considered from L_{j-1} are exactly those nodes containing v_j in their associated state. Doing so limits the number of merging operations that need to be performed, and it was computationally superior to other ordering heuristics as presented in Chapter 7.

4.6.3 Bounds vs. Maximum BDD Width

The purpose of this experiment is to analyze the impact of the maximum BDD width on the resulting bound. Figure 4.7 presents the results for instance `p-hat_300-1` in the `dimacs` set. The results are similar for other instances. The maximum width ranges from $W = 5$ to the value necessary to obtain the optimal value of 8. The bound approaches the optimal value almost monotonically as W increases, but the convergence is superexponential in W.

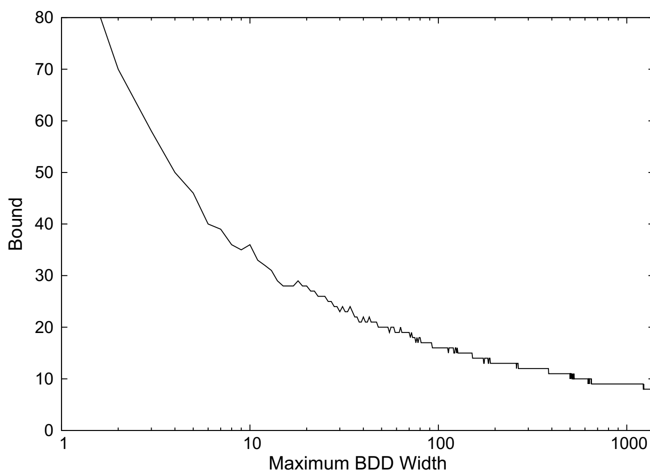


Fig. 4.7 Relaxation bound vs. maximum BDD width for the `dimacs` instance `p-hat_300-1`.

4.6.4 Comparison with LP Relaxation

We now address the key question of how BDD bounds compare with bounds produced by traditional LP relaxation and cutting planes. To obtain a tight initial LP relaxation, we used a *clique cover* model [78] of the maximum independent set problem, which requires computing a clique cover before the model can be formulated. We then augmented the LP relaxation with cutting planes generated at the root node by the CPLEX MILP solver.

Given a collection $\mathscr{C} \subseteq 2^V$ of cliques whose union covers all the edges of the graph G, the clique cover formulation is

$$\begin{aligned} \max \ & \sum_{v \in V} x_v \\ \text{s.t.} \ & \sum_{v \in S} x_v \leq 1, \ \text{for all } S \in \mathscr{C} \\ & x_v \in \{0,1\}. \end{aligned}$$

The clique cover $\mathscr{C}$ was computed using a greedy procedure. Starting with $\mathscr{C} = \emptyset$, let clique S consist of a single vertex v with the highest positive degree in G. Add to S the vertex with highest degree in $G \setminus S$ that is adjacent to all vertices in S, and repeat until no more additions are possible. At this point, add S to $\mathscr{C}$, remove from G all the edges of the clique induced by S, update the vertex degrees, and repeat the overall procedure until G has no more edges.

We solved the LP relaxation with ILOG CPLEX 12.4. We used the interior point (barrier) option because we found it to be up to 10 times faster than simplex on the larger LP instances. To generate cutting planes, we ran the CPLEX MILP solver with instructions to process the root node only. We turned off presolve, because no presolve is used for the BDD method, and it had only a marginal effect on the results in any case. Default settings were used for cutting plane generation.

The results for the `random` instance set appear in Table 4.1 and are plotted in Fig. 4.8. The table displays geometric means, rather than averages, to reduce the effect of outliers. It uses shifted geometric means[1] for computation times. The computation times for LP include the time necessary to compute the clique cover, which is much less than the time required to solve the initial LP for `random` instances, and about the same as the LP solution time for `dimacs` instances.

The results show that BDDs with width as small as 100 provide bounds that, after taking means, are superior to LP bounds for all graph densities except 0.1. The computation time required is about the same overall—more for sparse instances, less for dense instances. The scatter plot in Fig. 4.9 (top) shows how the bounds compare on individual instances. The fact that almost all points lie below the diagonal indicates the superior quality of BDD bounds.

Table 4.1 Bound quality and computation times for LP and BDD relaxations, using **random** instances. The bound quality is the ratio of the bound to the optimal value. The BDD bounds correspond to maximum BDD widths of 100, 1000, and 10,000. Each graph density setting is represented by 20 problem instances.

	Bound quality (geometric mean)					Time in seconds (shifted geom. mean)				
	LP relaxation		*BDD relaxation*			*LP relaxation*		*BDD relaxation*		
Density	LP only	LP+cuts	100	1,000	10,000	LP only	LP+cuts	100	1,000	10,000
0.1	1.60	1.50	1.64	1.47	1.38	0.02	3.74	0.13	1.11	15.0
0.2	1.96	1.76	1.80	1.55	1.40	0.04	9.83	0.10	0.86	13.8
0.3	2.25	1.93	1.83	1.52	1.40	0.04	7.75	0.08	0.82	11.8
0.4	2.42	2.01	1.75	1.37	1.17	0.05	10.6	0.06	0.73	7.82
0.5	2.59	2.06	1.60	1.23	1.03	0.06	13.6	0.05	0.49	3.88
0.6	2.66	2.04	1.43	1.10	1.00	0.06	15.0	0.04	0.23	0.51
0.7	2.73	1.98	1.28	1.00	1.00	0.07	15.3	0.03	0.07	0.07
0.8	2.63	1.79	1.00	1.00	1.00	0.07	9.40	0.02	0.02	0.02
0.9	2.53	1.61	1.00	1.00	1.00	0.08	4.58	0.01	0.01	0.01
All	**2.34**	**1.84**	**1.45**	**1.23**	**1.13**	**0.05**	**9.15**	**0.06**	**0.43**	**2.92**

[1] The shifted geometric mean of the quantities $v_1,\ldots,v_n$ is $g-\alpha$, where g is the geometric mean of $v_1+\alpha,\ldots,v_n+\alpha$. We used $\alpha = 1$ second.

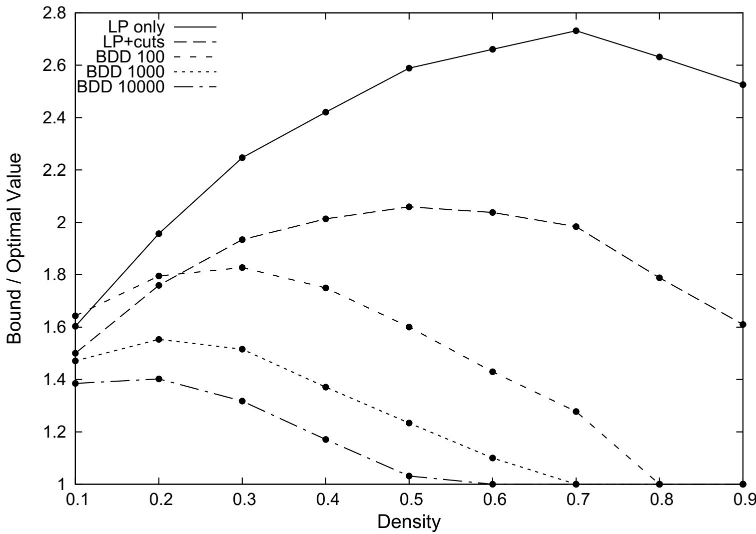


Fig. 4.8 Bound quality vs. graph density for `random` instances, showing results for LP only, LP plus cutting planes, and BDDs with maximum width 100, 1,000, and 10,000. Each data point is the geometric mean of 20 instances.

More important, however, is the comparison with the tighter bounds obtained by an LP with cutting planes, because this is the approach used in practice. BDDs of width 100 yield better bounds overall than even an LP with cuts, and they do so in less than 1% of the time. However, the mean bounds are worse for the two sparsest instance classes. By increasing the BDD width to 1000, the mean BDD bounds become superior for all densities, and they are still obtained in 5% as much time overall. See also the scatter plot in Fig. 4.9 (middle). Increasing the width to 10,000 yields bounds that are superior for every instance, as revealed by the scatter plot in Fig. 4.9 (bottom). The time required is about a third as much as LP overall, but somewhat more for sparse instances.

The results for the `dimacs` instance set appear in Table 4.2 and Fig. 4.10, with scatter plots in Fig. 4.11. The instances are grouped into five density classes, with the first class corresponding to densities in the interval $[0,0.2)$, the second class to the interval $[0.2,0.4)$, and so forth. The table shows the average density of each class. Table 4.3 shows detailed results for each instance.

BDDs of width 100 provide somewhat better bounds than the LP without cuts, except for the sparsest instances, and the computation time is somewhat less overall. Again, however, the more important comparison is with LP augmented by cutting planes. BDDs of width 100 are no longer superior, but increasing the width to

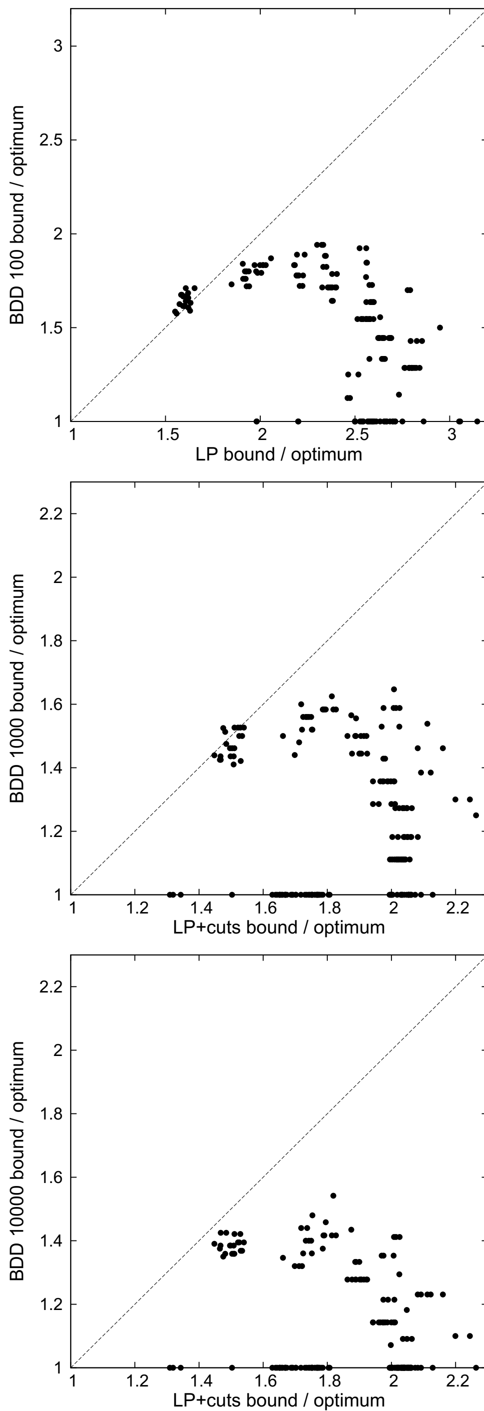


Fig. 4.9 Bound quality for an LP relaxation vs. width BDDs for `random` instances. Each data point represents one instance. The three plots show results for BDDs of maximum width 100 (top), 1000 (middle), and 10,000 (bottom). The LP bound in the last two plots benefits from cutting planes.

Table 4.2 Bound quality and computation times for LP and BDD relaxations, using `dimacs` instances. The bound quality is the ratio of the bound to the optimal value. The BDD bounds correspond to maximum BDD widths of 100, 1,000, and 10,000.

		Bound quality (geometric mean)					Time in seconds (shifted geom. mean)				
Avg.		*LP relaxation*		*BDD relaxation*			*LP relaxation*		*BDD relaxation*		
Density	Count	LP only	LP+cuts	100	1,000	10,000	LP only	LP+cuts	100	1,000	10,000
0.09	25	1.35	1.23	1.62	1.48	1.41	0.53	6.87	1.22	6.45	55.4
0.29	28	2.07	1.77	1.94	1.63	1.46	0.55	50.2	0.48	3.51	34.3
0.50	13	2.54	2.09	2.16	1.81	1.59	4.63	149	0.99	6.54	43.6
0.72	7	3.66	2.46	1.90	1.40	1.14	2.56	45.1	0.36	2.92	10.4
0.89	5	1.07	1.03	1.00	1.00	1.00	0.81	4.19	0.01	0.01	0.01
All	**78**	**1.88**	**1.61**	**1.78**	**1.54**	**1.40**	**1.08**	**27.7**	**0.72**	**4.18**	**29.7**

1000 yields better mean bounds than LP for all but the sparsest class of instances. The mean time required is about 15% that required by LP. Increasing the width to 10,000 yields still better bounds and requires less time for all but the sparsest instances. However, the mean BDD bound remains worse for instances with density less than 0.2. We conclude that BDDs are generally faster when they provide better bounds, and they provide better bounds, in the mean, for all but the sparsest `dimacs` instances.

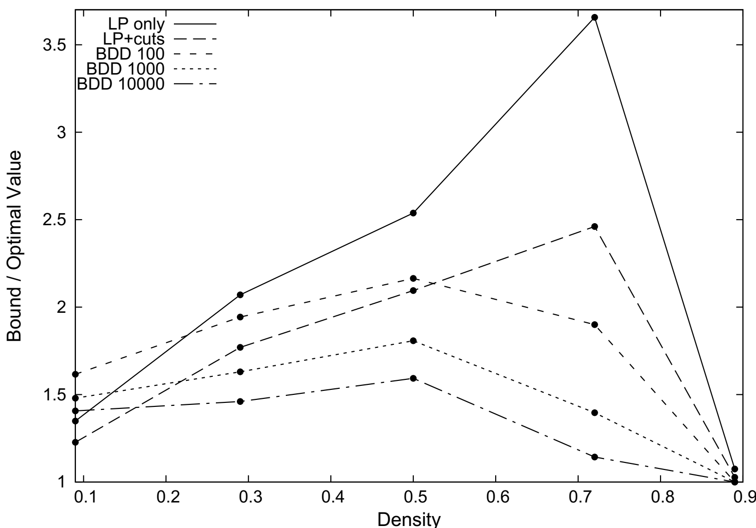


Fig. 4.10 Bound quality vs. graph density for `dimacs` instances, showing results for LP only, LP plus cutting planes, and BDDs with maximum width 100, 1,000, and 10,000. Each data point is the geometric mean of instances in a density interval of width 0.2.

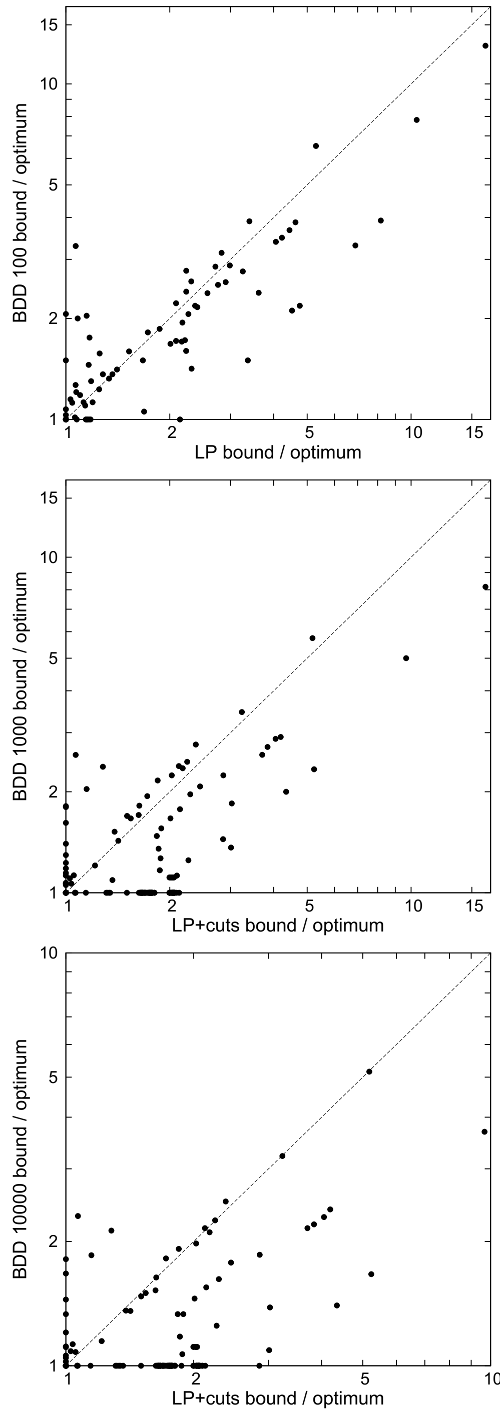


Fig. 4.11 Bound quality for an LP relaxation vs. BDDs for `dimacs` instances. The three plots show results for BDDs of maximum width 100 (top), 1000 (middle), and 10,000 (bottom). The LP bound in the last two plots benefits from cutting planes.

Table 4.3 Bound comparison for the `dimacs` instance set, showing the optimal value (Opt), the number of vertices (Size), and the edge density (Den). LP times correspond to clique cover generation (Clique), processing at the root node (CPLEX), and total time. The bound (Bnd) and computation time are shown for each BDD width. The best bounds are shown in boldface (either LP bound or one or more BDD bounds).

Instance				LP with cutting planes				Relaxed BDD					
					Time (sec)			Width 100		Width 1,000		Width 10,000	
Name	Opt	Size	Den	Bound	Clique	CPLEX	Total	Bnd	Sec	Bnd	Sec	Bnd	Sec
brock200_1	21	200	0.25	38.51	0	9.13	9.13	**36**	0.08	**31**	0.78	**28**	13.05
brock200_2	12	200	0.50	22.45	0.02	13.56	13.58	**17**	0.06	**14**	.45	**12**	4.09
brock200_3	15	200	0.39	28.20	0.01	11.24	11.25	**24**	0.06	**19**	0.70	**16**	8.31
brock200_4	17	200	0.34	31.54	0.01	9.11	9.12	**29**	0.08	**23**	0.81	**20**	10.92
brock400_1	27	400	0.25	66.10	0.05	164.92	164.97	68	0.34	**56**	3.34	**48**	47.51
brock400_2	29	400	0.25	66.47	0.04	178.17	178.21	69	0.34	**57**	3.34	**47**	51.44
brock400_3	31	400	0.25	66.35	0.05	164.55	164.60	67	0.34	**55**	3.24	**48**	47.29
brock400_4	33	400	0.25	66.28	0.05	160.73	160.78	68	0.35	**55**	3.32	**48**	47.82
brock800_1	23	800	0.35	96.42	0.73	1814.64	1815.37	**89**	1.04	**67**	13.17	**55**	168.72
brock800_2	24	800	0.35	97.24	0.73	1824.55	1825.28	**88**	1.02	**69**	13.11	**55**	180.45
brock800_3	25	800	0.35	95.98	0.72	2587.85	2588.57	**87**	1.01	**68**	12.93	**55**	209.72
brock800_4	26	800	0.35	96.33	0.73	1850.77	1851.50	**88**	1.02	**67**	12.91	**56**	221.07
C1000.9	68	1000	0.10	219.934	0.2	1204.41	1204.61	265	3.40	235	28.93	**219**	314.99
C125.9	34	125	0.10	41.29	0.00	1.51	1.51	45	0.05	**41**	0.43	**39**	5.73
C2000.5	16	2000	0.50	154.78	35.78	3601.41	3637.19	**125**	4.66	**80**	67.71	**59**	1207.69
C2000.9	77	2000	0.10	398.924	2.88	3811.94	3814.82	503	13.56	442	118.00	**397**	1089.96
C250.9	44	250	0.10	71.53	0.00	6.84	6.84	80	0.21	75	1.80	**67**	23.69
C4000.5	18	4000	0.50	295.67	631.09	3601.22	4232.31	**234**	18.73	**147**	195.05	**107**	3348.65
C500.9	57	500	0.10	124.21	0.03	64.56	64.59	147	0.85	134	7.42	**120**	84.66
c-fat200-1	12	200	0.92	**12.00**	0.04	0.95	0.99	**12**	0.00	**12**	0.00	**12**	0.00
c-fat200-2	24	200	0.84	**24.00**	0.05	0.15	0.2	**24**	0.00	**24**	0.00	**24**	0.00
c-fat200-5	58	200	0.57	61.70	0.07	35.85	35.92	**58**	0.00	**58**	0.00	**58**	0.00
c-fat500-10	126	500	0.63	**126.00**	1.89	2.80	4.69	**126**	0.01	**126**	0.01	**126**	0.01
c-fat500-1	14	500	0.96	16.00	1.03	27.79	28.82	**14**	0.02	**14**	0.01	**14**	0.01
c-fat500-2	26	500	0.93	**26.00**	0.81	7.71	8.52	**26**	0.01	**26**	0.00	**26**	0.01
c-fat500-5	64	500	0.81	**64.00**	1.51	3.05	4.56	**64**	0.01	**64**	0.01	**64**	0.01
gen200_p0.9_44	44	200	0.10	**44.00**	0.00	0.52	0.52	64	0.14	57	1.17	53	15.94
gen200_p0.9_55	55	200	0.10	**55.00**	0.00	2.04	2.04	65	0.14	63	1.19	61	15.74
gen400_p0.9_55	55	400	0.10	**55.00**	0.02	1.97	1.99	110	0.56	99	4.76	92	59.31
gen400_p0.9_65	65	400	0.10	**65.00**	0.02	3.08	3.1	114	0.55	105	4.74	94	56.99
gen400_p0.9_75	75	400	0.10	**75.00**	0.02	7.94	7.96	118	0.54	105	4.64	100	59.41
hamming10-2	512	1024	0.01	**512.00**	0.01	0.22	0.23	549	5.05	540	48.17	542	484.66
hamming10-4	40	1024	0.17	**51.20**	0.50	305.75	306.25	111	3.10	95	30.93	85	322.94
hamming6-2	32	64	0.10	**32.00**	0.00	0.00	0.00	**32**	0.01	**32**	0.09	**32**	1.20
hamming6-4	4	64	0.65	5.33	0.00	0.10	0.10	**4**	0.00	**4**	0.00	**4**	0.00
hamming8-2	128	256	0.03	**128.00**	0.00	0.01	0.01	132	0.26	136	2.45	131	25.70
hamming8-4	16	256	0.36	**16.00**	0.02	2.54	2.56	24	0.10	18	1.01	**16**	10.32
johnson16-2-4	8	120	0.24	**8.00**	0.00	0.00	0.00	12	0.02	**8**	0.10	**8**	0.23
johnson32-2-4	16	496	0.12	**16.00**	0.01	0.00	0.01	33	0.72	29	6.10	29	50.65
johnson8-2-4	4	28	0.44	**4.00**	0.00	0.00	0.00	**4**	0.00	**4**	0.00	**4**	0.00
johnson8-4-4	14	70	0.23	**14.00**	0.00	0.00	0.00	**14**	0.00	**14**	0.06	**14**	0.36
keller4	11	171	0.35	15.00	0.00	0.45	0.45	15	0.05	**12**	0.30	**11**	2.59
keller5	27	776	0.25	**31.00**	0.36	39.66	40.02	55	1.53	55	16.96	50	178.04
keller6	59	3361	0.18	**63.00**	55.94	3601.09	3657.03	194	37.02	152	361.31	136	3856.53
MANN_a27	126	378	0.01	**132.82**	0.00	1.31	1.31	152	0.46	142	3.71	136	41.90
MANN_a45	345	1035	0.00	**357.97**	0.01	1.47	1.48	387	2.83	367	26.73	389	285.05
MANN_a81	1100	3321	0.00	**1129.57**	0.07	11.22	11.29	1263	20.83	1215	254.23	1193	2622.59
MANN_a9	16	45	0.07	17.00	0.00	0.01	0.01	18	0.00	**16**	0.00	**16**	0.00
p_hat1000-1	10	1000	0.76	43.45	5.38	362.91	368.29	**33**	0.76	**20**	13.99	**14**	117.45
p_hat1000-2	46	1000	0.51	93.19	3.30	524.82	528.12	118	1.23	103	16.48	**91**	224.92
p_hat1000-3	68	1000	0.26	**152.74**	1.02	1112.94	1113.96	194	2.20	167	21.96	153	313.71
p_hat1500-1	12	1500	0.75	62.83	21.71	1664.41	1686.12	**47**	2.26	**28**	35.87	**20**	453.13
p_hat1500-2	65	1500	0.49	**138.13**	13.42	1955.38	1968.80	187	3.11	155	36.76	140	476.65
p_hat1500-3	94	1500	0.25	**223.60**	4.00	2665.67	2669.67	295	5.14	260	47.90	235	503.55
p_hat300-1	8	300	0.76	16.778	0.10	20.74	20.84	**12**	0.06	**9**	0.19	**8**	0.22
p_hat300-2	25	300	0.51	34.60	0.06	29.73	29.79	42	0.11	38	1.25	**34**	11.79
p_hat300-3	36	300	0.26	55.49	0.02	25.50	25.52	67	0.20	60	2.15	**54**	27.61
p_hat500-1	9	500	0.75	25.69	0.52	42.29	42.81	**19**	0.18	**13**	2.12	**9**	9.54
p_hat500-2	36	500	0.50	54.17	0.30	195.59	195.89	70	0.31	61	4.23	**53**	51.57
p_hat500-3	50	500	0.25	**86.03**	0.11	289.12	289.23	111	0.55	97	5.97	91	85.50
p_hat700-1	11	700	0.75	533.10	1.64	115.55	117.19	**24**	0.35	**15**	5.95	**12**	34.68
p_hat700-2	44	700	0.50	**71.83**	1.00	460.58	461.58	96	0.60	80	8.09	72	82.10
p_hat700-3	62	700	0.25	**114.36**	0.30	646.96	647.26	149	1.08	134	11.32	119	127.37
san1000	15	1000	0.50	16.00	43.14	180.46	223.60	19	1.14	**15**	15.01	**15**	99.71
san200_0.7_1	30	200	0.30	**30.00**	0.02	0.74	0.76	**30**	0.08	**30**	0.62	**30**	7.80
san200_0.7_2	18	200	0.30	**18.00**	0.02	1.55	1.57	19	0.06	**18**	0.50	**18**	6.50
san200_0.9_1	70	200	0.10	**70.00**	0.00	0.16	0.16	71	0.13	**70**	1.08	**70**	12.88
san200_0.9_2	60	200	0.10	**60.00**	0.00	0.49	0.49	66	0.13	**60**	1.14	**60**	14.96
san200_0.9_3	44	200	0.10	**44.00**	0.00	0.46	0.46	60	0.13	54	1.18	49	15.41
san400_0.5_1	13	400	0.50	**13.00**	1.09	10.08	11.17	**13**	0.19	**13**	1.27	**13**	5.00
san400_0.7_1	40	400	0.30	**40.00**	0.33	16.91	17.24	45	0.32	**40**	2.97	**40**	33.58
san400_0.7_2	30	400	0.30	**30.00**	0.31	12.22	12.53	39	0.32	32	3.50	**30**	38.96
san400_0.7_3	22	400	0.30	**22.00**	0.28	6.38	6.66	31	0.31	26	3.68	23	41.45
san400_0.9_1	100	400	0.10	**100.00**	0.02	6.52	6.54	123	0.56	107	4.66	**100**	57.46
sanr200_0.7	18	200	0.30	34.02	0.01	9.00	9.01	**31**	0.08	**28**	0.82	**24**	11.88
sanr200_0.9	42	200	0.10	59.60	0.00	3.32	3.32	67	0.14	60	1.17	**57**	15.51
sanr400_0.5	13	400	0.50	39.30	0.13	281.21	281.34	**31**	0.21	**24**	4.09	**18**	35.88
sanr400_0.7	21	400	0.30	60.05	0.06	168.64	168.70	**58**	0.30	**47**	3.52	**39**	51.93

4.7 Compiling Relaxed Diagrams by Separation

An alternative procedure to compile relaxed DDs can be obtained by modifying the separation procedure of Section 3.11 in a straightforward way. Recall that such a procedure would separate constraint classes one at a time by splitting nodes and removing arcs until the exact DD was attained. At each iteration of the separation procedure, the set of solutions represented in the DD was a superset of the solutions of the problem, and no feasible solutions were ever removed. Thus, the method already maintains a relaxed DD at all iterations (considering transition costs were appropriately assigned). To generate a limited-size relaxed DD, we could then simply stop the procedure when the size of a layer reached a given maximum width and output the current DD.

Even though valid, this method generates very weak relaxed DDs as not all constraints of the problem are necessarily considered in the relaxation, i.e., the procedure may stop if separation on the first constraints generates DDs with maximum width. A modified and more effective version of the separation procedure was developed in [84] and [94] under the name of *incremental refinement*. Incremental refinement was particularly used to create discrete relaxations for constraint satisfaction systems. As in the separation construction for exact DDs, the method also considers one constraint at a time. However, for each constraint, the steps of the algorithm are partitioned into two phases: *filtering* and *refinement*. Filtering consists of removing arcs for which all paths that cross them necessarily violate the constraint. Thus, in our notation, filtering is equivalent to removing arcs for which the corresponding transition functions lead to an infeasible state $\hat{0}$. Refinement consists of splitting nodes to strengthen the DD representation, as long as the size of the layer does not exceed the maximum width W. As before, we can split nodes based on the state associated with the constraint.

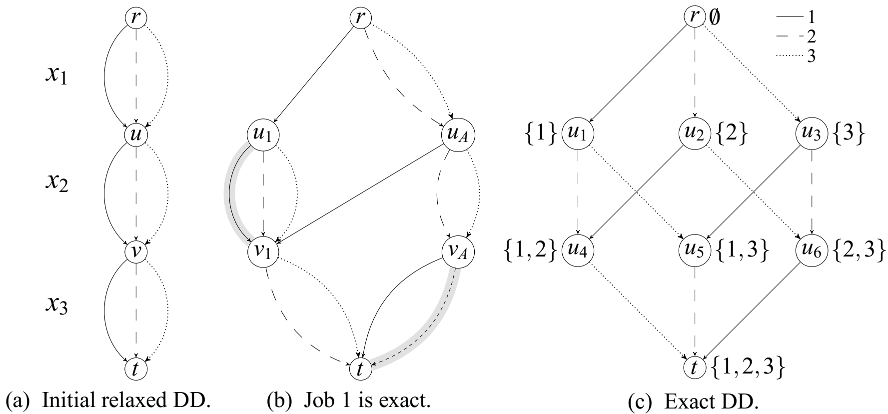
A key aspect of the filtering and refinement division is that both operations are perceived as independent procedures that can be modified or applied in any order that is suitable to the problem at hand. Even if the maximum width is already met, we can still apply the filtering operation of all constraints to remove infeasible arcs and strengthen the relaxation. Refinement may also be done in a completely heuristic fashion, or restricted to only some of the constraints of the problem. Moreover, we can introduce *redundant* states during filtering in order to identify sufficient conditions for the infeasibility of arcs, very much like redundant constraints in constraint programming potentially result in extra filtering of the

variable domains. Nevertheless, since not all nodes are split, their associated state may possibly represent an aggregation of several states from the exact DD. Extra care must be taken when defining the transition and cost functions to ensure the resulting DD is indeed a relaxation. This will be exemplified in Section 4.7.1.

A general outline of the relaxed DD compilation procedure is presented in Algorithm 4. The algorithm also requires a relaxed DD B' as input, which can be trivially obtained, e.g., using a 1-width DD as depicted in Fig. 4.12(a). The algorithm traverses the relaxed DD B' in a top-down fashion. For each layer j, the algorithm first performs filtering, i.e., it removes the infeasible arcs by checking whether the state transition function evaluates to an infeasible state $\hat{0}$. Next, the algorithm splits the nodes when the maximum width has not been met. If that is not the case, the procedure updates the state s associated with a node to ensure that the resulting DD is indeed a relaxation. Notice that the compilation algorithm is similar to Algorithm 2, except for the width limit, the order in which the infeasible state $\hat{0}$

Algorithm 4 Relaxed DD Compilation by Separation (Incremental Refinement): Max Width W

1: Let $B' = (U', A')$ be a DD such that $\text{Sol}(B') \supseteq \text{Sol}(\mathscr{P})$
2: **while** $\exists$ constraint C violated by B' **do**
3:    Let $s(u) = \chi$ for all nodes $u \in B'$
4:    $s(r) := \hat{r}$
5:    **for** $j = 1$ to n **do**
6:       *// Filtering*
7:       **for** $u \in L_j$ **do**
8:          **for** each arc $a = (u, v)$ leaving node u **do**
9:             **if** $t_j^C(s(u), d(a)) \neq \hat{0}$ **then**
10:                Remove arc a from B
11:       *// Refinement*
12:       **for** $u \in L_j$ **do**
13:          **for** each arc $a = (u, v)$ leaving node u **do**
14:             **if** $s(v) = \chi$ **then**
15:                $s(v) = t_j^C(s(u), d(a))$
16:             **else if** $s(v) \neq t_j^C(s(u), d(a))$ **and** $|L_j| < W$ **then**
17:                Remove arc (u, v)
18:                Create new node v' with $s(v') = t_j^C(u, d(a))$
19:                Add arc (u, v')
20:                Copy outgoing arcs from v as outgoing arcs from v'
21:                $L_j := L_j \cup \{v'\}$
22:             **else**
23:                Update $s(v)$ with $t_j^C(s(u), d(a))$



(a) Initial relaxed DD. (b) Job 1 is exact. (c) Exact DD.

Fig. 4.12 Three phases of refinement for a set of jobs $\{1,2,3\}$. Jobs are ranked lexicographically.

and the equivalence of states are checked, and the state update procedure. Filtering and refinement details (such as their order) can also be modified if appropriate.

4.7.1 Single-Machine Makespan Minimization

We now present an example of the incremental refinement procedure for the *single-machine makespan minimization problem* (MMP) presented in Section 3.8. Given a positive integer n, let $\mathscr{J} = \{1,\dots,n\}$ be a set of jobs that we wish to schedule on a machine that can process at most one job at a time. With each job we associate a processing time p_{ij}, indicating the time that job j requires on the machine if it is the i-th job to be processed. We wish to minimize the makespan of the schedule, i.e., the total completion time. As discussed in Section 3.8, the MMP can be written as the following optimization problem:

$$\begin{aligned} \min \ & \sum_{i=1}^{n} p_{i,x_i} \\ & \text{ALLDIFFERENT}(x_1,\dots,x_n) \\ & x_i \in \{1,\dots,n\}, \ i=1,\dots,n. \end{aligned} \tag{4.3}$$

We will now show how to define the filtering and refinement operations for the constraint (4.3). The feasible solutions are defined by all vectors x that satisfy the ALLDIFFERENT constraint in (4.3); that is, they are the permutations of J without

repetition. The states used for filtering and refinement for ALLDIFFERENT were initially introduced by [4] and [94].

4.7.1.1 Filtering

In the filtering operation we wish to identify conditions that indicate when all orderings identified by paths crossing an arc a always assign some job more than once. Let an arc a be *infeasible* if such a condition holds. We can directly use the state and transition function defined in Section 3.8; i.e., the state at a stage j represents the jobs already performed up to j. However, to strengthen the infeasibility test, we will also introduce an additional redundant state that provides a sufficient condition to remove arcs. This state represents *the jobs that might have been performed* up to a stage. To facilitate notation, we will consider a different state label $s(u)$ with each one of these states, as they can be computed simultaneously during the top-down procedure of Algorithm 4.

Namely, let us associate two states $All_u^{\downarrow} \subseteq \mathscr{J}$ and $Some_u^{\downarrow} \subseteq \mathscr{J}$ to each node u of the DD. The state $All_u^{\downarrow}$ is the set of arc labels that appear in *all* paths from the root node $\mathbf{r}$ to u (i.e., the same as in Section 3.8), while the state $Some_u^{\downarrow}$ is the set of arc labels that appear in *some* path from the root node $\mathbf{r}$ to u. We trivially have $All_{\mathbf{r}}^{\downarrow} = Some_{\mathbf{r}}^{\downarrow} = \emptyset$.

Instead of defining the transitions in functional form, we equivalently write them with respective to the graphical structure of the DD. To this end, let $in(v)$ be the set of incoming arcs at a node v. It follows from the definitions that $All_v^{\downarrow}$ and $Some_v^{\downarrow}$ for some node $v \neq \mathbf{r}$ can be recursively computed through the relations

$$All_v^{\downarrow} = \bigcap_{a=(u,v)\in in(v)} (All_u^{\downarrow} \cup \{d(a)\}), \tag{4.4}$$

$$Some_v^{\downarrow} = \bigcup_{a=(u,v)\in in(v)} (Some_u^{\downarrow} \cup \{d(a)\}). \tag{4.5}$$

For example, in Fig. 4.12(b) we have $All_{v_1}^{\downarrow} = \{1\}$ and $Some_u^{\downarrow} = \{1,2,3\}$.

Lemma 4.2. *An arc $a = (u,v)$ is infeasible if any of the following conditions holds:*

$$d(a) \in All_u^{\downarrow}, \tag{4.6}$$

$$|Some_u^{\downarrow}| = \ell(a) \quad \text{and} \quad d(a) \in Some_u^{\downarrow}. \tag{4.7}$$

Proof. The proof argument follows from [4]. Let π' be any partial ordering identified by a path from **r** to u that does not assign any job more than once. In condition (4.6), $d(a) \in All_u^{\downarrow}$ indicates that $d(a)$ is already assigned to some position in π', therefore appending the arc label $d(a)$ to π' will necessarily induce a repetition. For condition (4.7), notice first that the paths from **r** to u are composed of $\ell(a)$ arcs, and therefore π' represents an ordering with $\ell(a)$ positions. If $|Some_u^{\downarrow}| = \ell(a)$, then any $j \in Some_u^{\downarrow}$ is already assigned to some position in π', hence appending $d(a)$ to π' also induces a repetition. □

Thus, the tests (4.6) and (4.7) can be applied in lines 6 to 10 in Algorithm 4 to remove infeasible arcs. For example, in Fig. 4.12(b) the two shaded arcs are infeasible. The arc (u_1, v_1) with label 1 is infeasible due to condition (4.6) since $All_{u_1}^{\downarrow} = \{1\}$. The arc (v_A, t) with label 2 is infeasible due to condition (4.7) since $2 \in Some_{v_A}^{\downarrow} = \{2,3\}$ and $|Some_{v_A}^{\downarrow}| = 2$.

We are also able to obtain stronger tests by equipping the nodes with additional states that can be derived from a *bottom-up* perspective of the DD. Namely, as in [94], we define two new states $All_u^{\uparrow} \subseteq \mathscr{J}$ and $Some_u^{\uparrow} \subseteq \mathscr{J}$ for each node u of $\mathscr{M}$. They are equivalent to the states $All_u^{\downarrow}$ and $Some_u^{\downarrow}$, but now they are computed with respect to the paths from **t** to u instead of the paths from **r** to u. As before, they are recursively obtained through the relations

$$All_u^{\uparrow} = \bigcap_{a=(u,v)\in out(u)} (All_v^{\uparrow} \cup \{d(a)\}), \tag{4.8}$$

$$Some_u^{\uparrow} = \bigcup_{a=(u,v)\in out(u)} (Some_v^{\uparrow} \cup \{d(a)\}), \tag{4.9}$$

which can be computed by a bottom-up breadth-first search before the top-down procedure.

Lemma 4.3. *An arc $a = (u,v)$ is infeasible if any of the following conditions holds:*

$$d(a) \in All_v^{\uparrow}, \tag{4.10}$$

$$|Some_v^{\uparrow}| = n - \ell(a) \quad \text{and} \quad d(a) \in Some_v^{\uparrow}, \tag{4.11}$$

$$|Some_u^{\downarrow} \cup \{d(a)\} \cup Some_v^{\uparrow}| < n. \tag{4.12}$$

Proof. The proofs for conditions (4.10) and (4.11) follow from an argument in [94] and are analogous to the proof of Lemma 4.2. Condition (4.12) implies that any ordering identified by a path containing a will never assign all jobs $\mathscr{J}$.

4.7.1.2 Refinement

Refinement consists of splitting nodes to remove paths that encode infeasible solutions, therefore strengthening the relaxed DD. Ideally, refinement should modify a layer so that each of its nodes exactly represents a particular state of each constraint. However, as it may be necessary to create an exponential number of nodes to represent all such states, some heuristic decision must be considered on which nodes to split in order to observe the maximum alloted width.

In this section we present a heuristic refinement procedure that exploits the structure of the ALLDIFFERENT constraint. Our goal is to be as precise as possible with respect to the jobs with a higher *priority*, where the priority of a job is defined according to the problem data. More specifically, we will develop a refinement heuristic that, when combined with the infeasibility conditions for the permutation structure, yields a relaxed MDD where the jobs with a high priority are represented exactly with respect to that structure; that is, these jobs are assigned to exactly one position in all orderings encoded by the relaxed MDD.

Thus, if higher priority is given to jobs that play a greater role in the feasibility or optimality of the problem at hand, the relaxed MDD may represent more accurately the feasible orderings of the problem, providing, e.g., better bounds on the objective function value. For example, if we give priority to jobs with a larger processing time, the bound on the makespan would be potentially tighter with respect to the ones obtained from other possible relaxed MDDs for this same instance. We will exploit this property for a number of scheduling problems in Chapter 11.

To achieve this property, the refinement heuristic we develop is based on the following theorem, which we will prove constructively later:

Theorem 4.3. *Let $W > 0$ be the maximum MDD width. There exists a relaxed MDD $\mathscr{M}$ where at least $\lfloor \log_2 W \rfloor$ jobs are assigned to exactly one position in all orderings identified by $\mathscr{M}$.*

Let us represent the job priorities by defining a *ranking* of jobs $\mathscr{J}^* = \{j_1^*, \dots, j_n^*\}$, where jobs with smaller index in $\mathscr{J}^*$ have a higher priority. We can thus achieve the desired property of our heuristic refinement by constructing the relaxed MDD $\mathscr{M}$ based on Theorem 4.3, where we ensure that the jobs exactly represented in $\mathscr{M}$ are those with a higher ranking.

Before proving Theorem 4.3, we first identify conditions on when a node violates the desired refinement property and needs to be modified. To this end, let $\mathscr{M}$ be any

relaxed MDD. Assume the states $All_u^{\downarrow}$ and $Some_u^{\downarrow}$ as described before are computed for all nodes u in $\mathscr{M}$, and no arcs satisfy the infeasibility conditions (4.6) to (4.12). We have the following lemma:

Lemma 4.4. *A job j is assigned to exactly one position in all orderings identified by $\mathscr{M}$ if and only if $j \notin Some_u^{\downarrow} \setminus All_u^{\downarrow}$ for all nodes $u \in \mathscr{M}$.*

Proof. Suppose first that a job j is assigned to exactly one position in all orderings identified by $\mathscr{M}$, and take a node u in $\mathscr{M}$ such that $j \in Some_u^{\downarrow}$. From the definition of $Some_u^{\downarrow}$, there exists a path from $\mathbf{r}$ to u with an arc labeled j. This implies by hypothesis that all paths from u to $\mathbf{t}$ do not have any arcs labeled j, otherwise we will have a path that identifies an ordering where j is assigned more than once. But then, also by hypothesis, all paths from $\mathbf{r}$ to u must necessarily have some arc labeled j, thus $j \in All_u^{\downarrow}$, which implies $j \notin Some_u^{\downarrow} \setminus All_u^{\downarrow}$.

Conversely, suppose $j \in Some_u^{\downarrow} \setminus All_u^{\downarrow}$ for all nodes u in $\mathscr{M}$. Then a node u can only have an outgoing arc a with $d(a) = j$ if $j \notin Some_u^{\downarrow}$, which is due to the filtering rule (4.6). Thus, no job is assigned more than once in any ordering encoded by $\mathscr{M}$. Finally, rule (4.12) ensures that j is assigned exactly once in all paths. □

We now provide a constructive proof for Theorem 4.3.

Proof. Proof of Theorem 4.3: Let $\mathscr{M}$ be a 1-width relaxation. We can obtain the desired MDD by applying filtering and refinement on $\mathscr{M}$ in a top-down approach as described in Section 4.7. For filtering, remove all arcs satisfying the infeasibility rules (4.6) and (4.7). For refining a particular layer L_i, apply the following procedure: For each job $j = j_1, \ldots, j_n$ in this order, select a node $u \in L_i$ such that $j \in Some_u^{\downarrow} \setminus All_u^{\downarrow}$. Create two new nodes u_1 and u_2, and redirect the incoming arcs at u to u_1 and u_2 as follows: if the arc $a = (v, u)$ is such that $j \in (All_v^{\downarrow} \cup \{d(a)\})$, redirect it to u_1; otherwise, redirect it to u_2. Replicate all the outgoing arcs of u to u_1 and u_2, remove u, and repeat this until the maximum width W is met, there are no nodes satisfying this for j, or all jobs were considered.

We now show that this refinement procedure suffices to produce a relaxed MDD satisfying the conditions of the theorem. Observe first that the conditions of Lemma 4.4 are satisfied by any job at the root node $\mathbf{r}$, since $Some_{\mathbf{r}}^{\downarrow} = \emptyset$. Suppose, by induction hypothesis, that the conditions of Lemma 4.4 are satisfied for some job j at all nodes in layers $L_1, \ldots, L_{i'}$, $i' < i$, and consider we created nodes u_1 and u_2 from some node $u \in L_i$ such that $j \in Some_u^{\downarrow} \setminus All_u^{\downarrow}$ as described above. By construction, any incoming arc $a = (v, u_2)$ at u_2 satisfies $j \notin (All_v^{\downarrow} \cup \{d(a)\})$; by induction hypothesis,

$j \notin Some^{\downarrow}_{v}$, hence $j \notin Some^{\downarrow}_{u_2} \setminus All^{\downarrow}_{u_2}$ by relation (4.4). Analogously, we can show $j \in All^{\downarrow}_{u_1}$, thus $j \notin Some^{\downarrow}_{u_1} \setminus All^{\downarrow}_{u_1}$.

Since the jobs $\mathscr{J}$ are processed in the same order for all layers, we just need now to compute the minimum number of jobs for which all nodes violating Lemma 4.4 were split when the maximum width W was attained. Just observe that, after all the nodes were verified with respect to a job, we at most duplicated the number of nodes in a layer (since each split produces one additional node). Thus, if m jobs were considered, we have at most 2^m nodes in a layer, thus at least $\lfloor \log_2 W \rfloor$ nodes will be exactly represented in $\mathscr{M}$. □

We can utilize Theorem 4.3 to guide our top-down approach for filtering and refinement, following the refinement heuristic based on the job ranking $\mathscr{J}^*$ described in the proof of Theorem 4.3. Namely, we apply the following refinement at a layer L_i: For each job $j^* = j^*_1, \ldots, j^*_n$ in the order defined by $\mathscr{J}^*$, identify the nodes u such that $j^* \in Some^{\downarrow}_{u} \setminus All^{\downarrow}_{u}$ and split them into two nodes u_1 and u_2, where an incoming arc $a = (v, u)$ is redirected to u_1 if $j^* \in (All^{\downarrow}_{v} \cup \{d(a)\})$ or u_2 otherwise, and replicate all outgoing arcs for both nodes. Moreover, if the relaxed MDD is a 1-width relaxation, then we obtain the bound guarantee on the number of jobs that are exactly represented.

This procedure also yields a *reduced* MDD $\mathscr{M}$ for certain structured problems, which we will show in Section 11.7. It provides sufficient conditions to split nodes for any problem where an ALLDIFFERENT constraint is stated on the variables. Lastly, recall that equivalence classes corresponding to constraints other than the permutation structure may also be taken into account during refinement. Therefore, if the maximum width W is not met in the refinement procedure above, we assume that we will further split nodes by arbitrarily partitioning their incoming arcs. Even though this may yield false equivalence classes, the resulting $\mathscr{M}$ is still a valid relaxation and may provide a stronger representation.

As an illustration, let $\mathscr{J} = \{1, 2, 3\}$ and assume jobs are ranked lexicographically. Given the relaxed DD in Fig. 4.12(a), Fig. 4.12(b) without the shaded arcs depicts the result of the refinement heuristics for a maximum width of 2. Notice that job 1 appears exactly once in all solutions encoded by the DD. Figure 4.12(c) depicts the result of the refinement for a maximum width of 3. It is also exact and reduced (which is always the case if we start with a 1-width relaxation and the constraint set is composed of only one ALLDIFFERENT).

Chapter 5
Restricted Decision Diagrams

Abstract This chapter presents a general-purpose methodology for obtaining a set of feasible solutions to a discrete optimization problems using *restricted decision diagrams*. A restricted diagram can be perceived as a counterpart of the concept of relaxed diagrams introduced in previous chapters, and represents an underapproximation of the feasible set, the objective function, or both. We first show how to modify the top-down compilation approach to generate restricted diagrams that observe an input-specified width. Next, we provide a computational study of the bound provided by restricted diagrams, particularly focusing on the set covering and set packing problems.

5.1 Introduction

General-purpose algorithms for discrete optimization are commonly branch-and-bound methods that rely on two fundamental components: a relaxation of the problem, such as a linear programming relaxation of an integer programming model, and primal heuristics. Heuristics are used to provide feasible solutions during the search for an optimal one, which in practice can be often more important than providing a proof of optimality.

Much of the research effort dedicated to developing heuristics for discrete optimization has primarily focused on specific combinatorial optimization problems. This includes, e.g., the set covering problem [39] and the maximum clique problem [77, 130]. In contrast, general-purpose heuristics have received much less attention in the literature. The vast majority of the general techniques are embodied in integer

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_5

programming technology, such as the *feasibility pump* [66] and the *pivot, cut, and dive* heuristic [60]. A survey of heuristics for integer programming is presented by [73, 74] and [30]. Local search methods for binary problems can also be found in [1] and [31].

In this chapter we present a general-purpose heuristic based on *restricted* decision diagrams (DDs). A weighted DD B is restricted for an optimization problem $\mathscr{P}$ if B represents a subset of the feasible solutions of $\mathscr{P}$, and path lengths are lower bounds on the value of feasible solutions. That is, B is restricted for $\mathscr{P}$ if

$$\mathrm{Sol}(\mathscr{P}) \supseteq \mathrm{Sol}(B), \tag{5.1}$$

$$f(x^p) \geq v(p), \text{ for all } r\text{–}t \text{ paths } p \text{ in } B \text{ for which } x^p \in \mathrm{Sol}(\mathscr{P}). \tag{5.2}$$

Suppose $\mathscr{P}$ is a maximization problem. In Chapter 3, we showed that an exact DD reduces discrete optimization to a longest-path problem: If p is a longest path in a BDD B that is exact for $\mathscr{P}$, then x^p is an optimal solution of $\mathscr{P}$, and its length $v(p)$ is the optimal value $z^*(\mathscr{P}) = f(x^p)$ of $\mathscr{P}$. When B is restricted for $\mathscr{P}$, a longest path p provides a *lower bound* on the optimal value. The corresponding solution x^p is *always* feasible and $v(p) \leq z^*(\mathscr{P})$. Hence, restricted DDs provide a primal solution to the problem. As in relaxed DDs, the width of a restricted DDs is limited by an input parameter, which can be adjusted according to the number of variables of the problem and computer resources.

For example, consider the graph and vertex weights depicted in Fig. 5.1 (the same as Fig. 4.1). Figure 5.2(a) represents an exact BDD in which each path corresponds to an independent set encoded by the arc labels along the path, and each independent set corresponds to some path. The longest r–t path in the BDD has value 11, corresponding to solution $x = (0,1,0,0,1)$ and to the independent set $\{2,5\}$, the maximum-weight independent set in the graph.

Figure 5.2(b) shows a restricted BDD for the same problem instance. Each path p in the BDD encodes a feasible solution x^p with length equal to the corresponding independent set weight. However, not all independent sets of G are encoded in the BDD, such as the optimal independent set $\{2,5\}$ for the original problem. The longest path in the restricted DD corresponds to solution $(1,0,0,0,1)$ and independent set $\{1,5\}$, and thus provides a lower bound of 10 on the objective function.

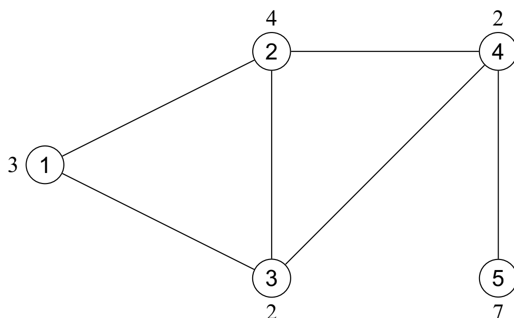


Fig. 5.1 Graph with vertex weights for the MISP.

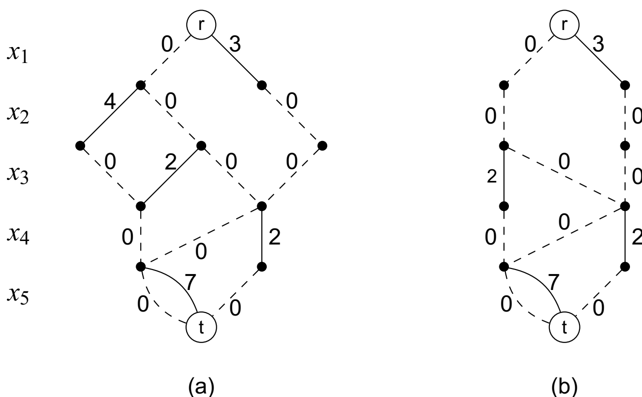


Fig. 5.2 (a) Exact BDD and (b) restricted BDD for the MISP on the graph in Fig. 5.1.

5.2 Top-Down Compilation of Restricted DDs

Restricted BDDs can be constructed in a much simpler way than relaxed DDs. We need only eliminate nodes from a layer when the layer becomes too large. Given a valid DP formulation of a discrete optimization problem $\mathscr{P}$ and a maximum width W, Algorithm 5 outputs a restricted DD for $\mathscr{P}$. Note that it is similar to Algorithm 1 except for lines 3 to 5. Condition (5.1) for a restricted BDD is satisfied because the algorithm only deletes solutions, and furthermore, since the algorithm never modifies the states of any nodes that remain, condition (5.2) must also be satisfied. Finally, nodes to be eliminated are also selected heuristically according to a pre-defined function *node_select*.

We remark in passing that it is also possible to apply Algorithm 3 to obtain restricted DDs. To this end, we modify the operator $\oplus(M)$ so that the algorithm

Algorithm 5 Restricted DD Top-Down Compilation for Maximum Width W

1: Create node $r = \hat{r}$ and let $L_1 = \{r\}$
2: **for** $j = 1$ to n **do**
3:    **while** $|L_j| > W$ **do**
4:       let $M = \text{node_select}(L_j)$
5:       $L_j \leftarrow (L_j \setminus M)$
6:    let $L_{j+1} = \emptyset$
7:    **for all** $u \in L_j$ and $d \in D(x_j)$ **do**
8:       **if** $t_j(u,d) \neq \hat{0}$ **then**
9:          let $u' = t_j(u,d)$, add u' to L_{j+1}, and set $b_d(u) = u'$, $v(u,u') = h_j(u,u')$
10: Merge nodes in L_{n+1} into terminal node t

outputs restrictions instead of relaxations. For example, in the MISP relaxation described in Section 4.3, we could apply the *intersection* operator as opposed to the union operator. Such a technique will not be exploited in this book, but it could be useful to ensure certain properties of the restricted DD (e.g., it can be shown that a restricted DD built with $\oplus(M)$ may contain more solutions than the one obtained by directly removing nodes).

5.3 Computational Study

In this section, we present a computational study on randomly generated set covering and set packing instances. The set covering problem (SCP) and the set packing problem (SPP) were first presented, as integer programming models, in Sections 3.6 and 3.7, respectively. We evaluate our method by comparing the bounds provided by a restricted BDD with the ones obtained via state-of-the-art integer programming (IP) technology. We acknowledge that a procedure solely geared toward constructing heuristic solutions is in principle favored against general-purpose IP solvers. Nonetheless, we sustain that this is still a meaningful comparison, as modern IP solvers are the best-known general bounding technique for 0–1 problems due to their advanced features and overall performance. This method of testing new heuristics for binary optimization problems was employed by the authors in [31], and we provide a similar study here to evaluate the effectiveness of our algorithm.

The DP models for the set covering and set packing problems are the ones described in Sections 3.6 and 3.7. The tests ran on an Intel Xeon E5345 with 8 GB of RAM. The BDD code was implemented in C++. We used ILOG CPLEX 12.4 as our

IP solver. In particular, we took the bound obtained from the root node relaxation. We set the solver parameters to balance the quality of the bound value and the CPU time to process the root node. The CPLEX parameters that are distinct from the default settings are presented in Table 5.1. We note that all cuts were disabled, since we observed that the root node would be processed orders of magnitude faster without adding cuts, which did not have a significant effect on the quality of the heuristic solution obtained for the instances tested.

Table 5.1 CPLEX parameters.

Parameters (CPLEX internal name)	*Value*
Version	12.4
Number of explored nodes (`NodeLim`)	0 (only root)
Parallel processes (`Threads`)	1
Cuts (`Cuts`, `Covers`, `DisjCuts`, ...)	-1 (off)
Emphasis (`MIPEmphasis`)	4 (find hidden feasible solutions)
Time limit (`TiLim`)	3600

Our experiments focus on instances with a particular structure. Namely, we provide evidence that restricted BDDs perform well when the constraint matrix has a small *bandwidth*. The bandwidth of a matrix A is defined as

$$b_w(A) = \max_{i \in \{1,2,\ldots,m\}} \{ \max_{j,k: a_{i,j}, a_{i,k}=1} \{ j - k \} \}.$$

The bandwidth represents the largest distance, in the variable ordering given by the constraint matrix, between any two variables that share a constraint. The smaller the bandwidth, the more structured the problem, in that the variables participating in common constraints are close to each other in the ordering. The *minimum bandwidth problem* seeks to find a variable ordering that minimizes the bandwidth [114, 51, 62, 80, 115, 127, 140]. This underlying structure, when present in A, can be captured by BDDs, resulting in good computational performance.

5.3.1 Problem Generation

Our random matrices are generated according to three parameters: the number of variables n, the number of ones per row k, and the bandwidth b_w. For a fixed n, k, and b_w, a random matrix A is constructed as follows: We first initialize A as a

zero matrix. For each row i, we assign the ones by selecting k columns uniformly at random from the index set corresponding to the variables $\{x_i, x_{i+1}, \ldots, x_{i+b_w}\}$. As an example, a constraint matrix with $n = 9$, $k = 3$, and $b_w = 4$ may look like

$$A = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

Consider the case when $b_w = k$. The matrix A has the *consecutive ones property* and is totally unimodular [69], and IP is able to find the optimal solution for the set packing and set covering instances at the root node. Similarly, we argue that an $(m+1)$-width restricted BDD is an exact BDD for both classes of problems, hence also yielding an optimal solution for when this structure is present. Indeed, we show that A having the consecutive ones property implies that the state of a BDD node u is always of the form $\{j, j+1, \ldots, m\}$ for some $j \geq L(u)$ during top-down compilation.

To see this, consider the set covering problem. For a partial solution x identified by a path from r to a certain node u in the BDD, let $\mathtt{s}(x)$ be the set covering state associated with u. We claim that, for any partial solution x' that can be completed to a feasible solution, $\mathtt{s}(x') = \{i(x'), i(x')+1, \ldots, m\}$ for some variable index $i(x')$, or $\mathtt{s}(x') = \emptyset$ if x' satisfies all of the constraints when completed with 0's. Let $j' \leq j$ be the largest index in x' with $x'_j = 1$. Because x' can be completed to a feasible solution, for each $i \leq b_w + j - 1$ there is a variable x_{j_i} with $a_{i,j_i} = 1$. All other constraints must have $x_j = 0$ for all i with $a_{i,j} = 0$. Therefore $\mathtt{s}(x') = \{b_w + j, b_w + j + 1, \ldots, m\}$, as desired. Hence, the state of every partial solution must be of the form $i, i+1, \ldots, m$ or $\emptyset$. Because there are at most $m+1$ such states, the size of any layer cannot exceed $(m+1)$. A similar argument works for the SPP.

Increasing the bandwidth b_w, however, destroys the totally unimodular property of A and the bounded width of B. Hence, by changing b_w, we can test how sensitive IP and the BDD-based heuristics are to the staircase structure dissolving.

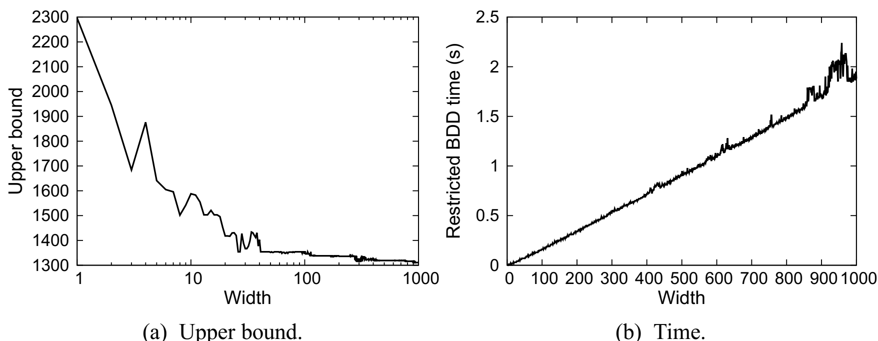
We note here that generating instances of this sort is not restrictive. Once the bandwidth is large, the underlying structure dissolves and each element of the matrix becomes randomly generated. In addition, as mentioned above, algorithms to solve the minimum bandwidth problem exactly or approximately have been investigated.

To any SCP or SPP one can therefore apply these methods to reorder the matrix and then apply the BDD-based algorithm.

5.3.2 Solution Quality and Maximum BDD Width

We first analyze the impact of the maximum width W on the solution quality provided by a restricted BDD. To this end, we report the generated bound versus the maximum width W obtained for a set covering instance with $n = 1,000$, $k = 100$, $b_w = 140$, and a cost vector c where each c_j was chosen uniformly at random from the set $\{1,\ldots,nc_j\}$, where nc_j is the number of constraints in which variable j participates. We observe that the reported results are common among all instances tested.

Figure 5.3(a) depicts the resulting bounds, where the width axis is on a logarithmic scale, and Fig. 5.3(b) presents the total time to generate the W-restricted BDD and extract its best solution. We tested all W in the set $\{1,2,3,\ldots,1,000\}$. We see that, as the width increases, the bound approaches the optimal value, with a super-exponential-like convergence in W. The time to generate the BDD grows linearly in W, which can be shown to be consistent with the complexity of the construction algorithm.



(a) Upper bound.

(b) Time.

Fig. 5.3 Restricted BDD performance versus the maximum allotted width for a set covering instance with $n = 1000$, $k = 100$, $b_w = 140$, and random cost vector.

5.3.3 Set Covering

First, we report the results for two representative classes of instances for the set covering problem. In the first class, we studied the effect of b_w on the quality of the bound. To this end, we fixed $n = 500$, $k = 75$, and considered b_w as a multiple of k, namely $b_w \in \{\lfloor 1.1k \rfloor, \lfloor 1.2k \rfloor, \dots, \lfloor 2.6k \rfloor\}$. In the second class, we analyzed if k, which is proportional to the density of A, also has an influence on the resulting bound. For this class we fixed $n = 500$, $k \in \{25, 50, \dots, 250\}$, and $b_w = 1.6k$. In all classes we generated 30 instances for each triple (n, k, b_w) and fixed 500 as the restricted BDD maximum width.

It is well known that the objective function coefficients play an important role in the bound provided by IP solvers for the set covering problem. We considered two types of cost vectors c in our experiments. The first is $c = \mathbf{1}$, which yields the *combinatorial* set covering problem. For the second cost function, let nc_j be the number of constraints that include variable x_j, $j = 1, \dots, n$. We chose the cost of variable x_j uniformly at random from the range $[0.75nc_j, 1.25nc_j]$. As a result, variables that participate in more constraints have a higher cost, thereby yielding harder set covering problems to solve. This cost vector yields the *weighted* set covering problem.

The feasible solutions are compared with respect to their *optimality gap*. The optimality gap of a feasible solution is obtained by first taking the absolute difference between its objective value and a lower bound to the problem, and then dividing this by the solution's objective value. In both BDD and IP cases, we used the dual value obtained at the root node of CPLEX as the lower bound for a particular problem instance.

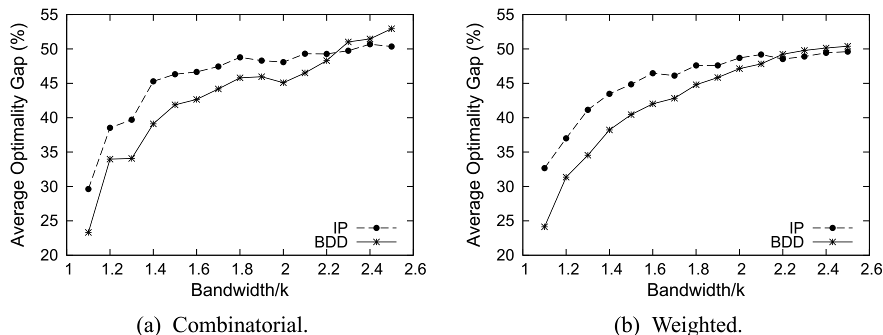
The results for the first instance class are presented in Fig. 5.4. Each data point in the figure represents the average optimality gap, over the instances with that configuration. We observe that the restricted BDD yields a significantly better solution for small bandwidths in the combinatorial set covering version. As the bandwidth increases, the staircase structure is lost and the BDD gap becomes progressively worse in comparison with the IP gap. This is a result of the increasing width of the exact reduced BDD for instances with larger bandwidth matrices. Thus, more information is lost when we restrict the BDD size. The same behavior is observed for the weighted set covering problem, although the gap provided by the restricted BDD is generally better than the IP gap even for larger bandwidths. Finally, we note that the restricted BDD time is also comparable to the IP time, which is on average

less than 1 second for this configuration. This time takes into account both BDD construction and extraction of the best solution it encodes by means of a shortest path algorithm.

The results for the second instance class are presented in Fig. 5.5. We note that restricted BDDs provide better solutions when k is smaller. One possible explanation for this behavior is that a sparser matrix causes variables to participate in fewer constraints, thereby decreasing the possible number of BDD node states. Again, less information is lost by restricting the BDD width. Moreover, we note once again that the BDD performance, when compared with CPLEX, is better for the weighted instances tested. Finally, we observe that the restricted BDD time is similar to the IP time, always below one second for instances with 500 variables.

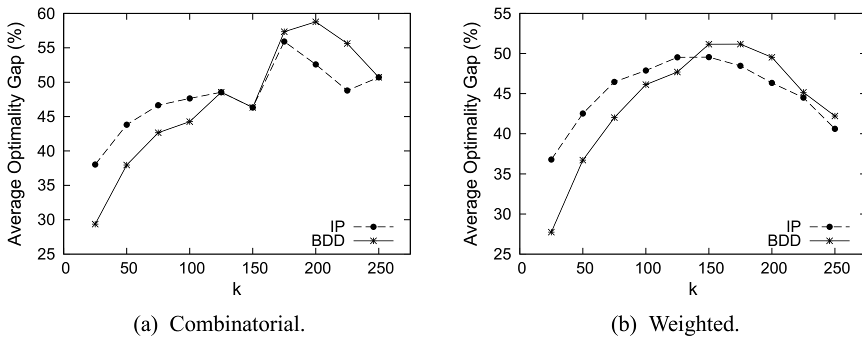
Next, we compare solution quality and time as the number of variables n increases. We generated random instances with $n \in \{250, 500, 750, \ldots, 4,000\}$, $k = 75$, and $b_w = 2.2k = 165$ to this end. The choice of k and b_w was motivated by Fig. 5.4(b), corresponding to the configuration where IP outperforms BDD with respect to solution quality when $n = 500$. As before, we generated 30 instances for each n. Moreover, only weighted set covering instances are considered in this case.

The average optimality gap and time are presented in Figs. 5.6(a) and 5.6(b), respectively. The y axis in Fig. 5.6(b) is on logarithmic scale. For $n > 500$, we observe that the restricted BDDs yield better-quality solutions than the IP method, and as n increases this gap remains constant. However, the IP times grow at a much faster rate than the restricted BDD times. In particular, with $n = 4,000$, the BDD times are approximately two orders of magnitude faster than the corresponding IP times.



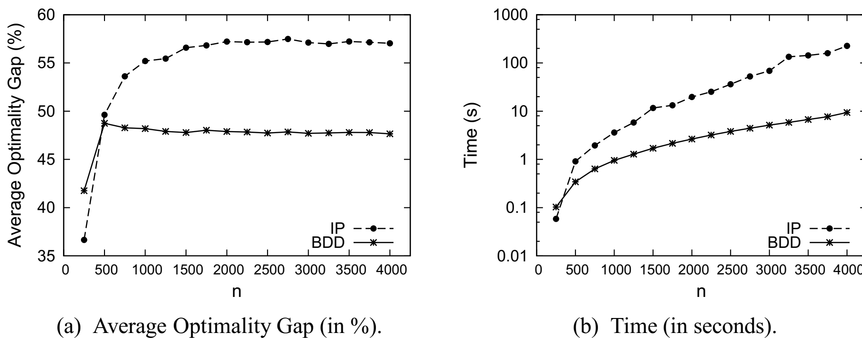
(a) Combinatorial. (b) Weighted.

Fig. 5.4 Average optimality gaps for combinatorial and weighted set covering instances with $n = 500$, $k = 75$, and varying bandwidth.



(a) Combinatorial. (b) Weighted.

Fig. 5.5 Average optimality gaps for combinatorial and weighted set covering instances with $n = 500$, varying k, and $b_w = 1.6k$.



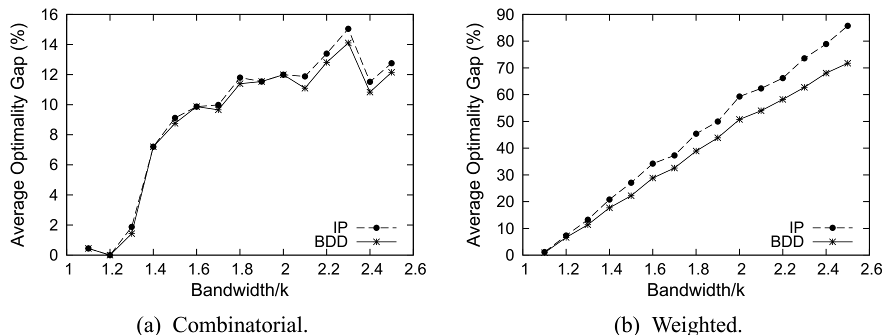
(a) Average Optimality Gap (in %). (b) Time (in seconds).

Fig. 5.6 Average optimality gaps and times for weighted set covering instances with varying n, $k = 75$, and $b_w = 2.2k = 165$. The y axis in the time plot is on logarithmic scale.

5.3.4 Set Packing

We extend the same experimental analysis of the previous section to set packing instances. Namely, we initially compare the quality of the solutions by means of two classes of instances. In the first class we analyze variations of the bandwidth by generating random instances with $n = 500$, $k = 75$, and setting b_w in the range $\{\lfloor 1.1k \rfloor, \lfloor 1.2k \rfloor, \ldots, \lfloor 2.5k \rfloor\}$. In the second class, we analyze variations in the density of the constraint matrix A by generating random instances with $n = 500$, $k \in \{25, 50, \ldots, 250\}$, and with a fixed $b_w = 1.6k$. In all classes, we created 30 instances for each triple (n, k, b_w) and set 500 as the restricted BDD maximum width.

The quality is also compared with respect to the optimality gap of the feasible solutions, which is obtained by dividing the absolute difference between the solution's



(a) Combinatorial.

(b) Weighted.

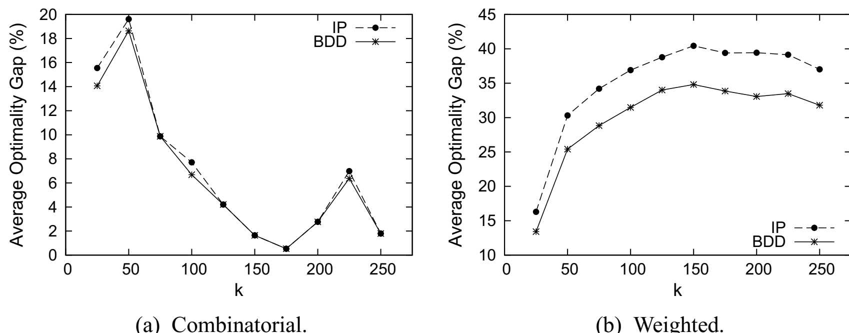
Fig. 5.7 Average optimality gaps for combinatorial and weighted set packing instances with $n = 500$, $k = 75$, and varying bandwidth.

objective value and an upper bound to the problem by the solution's objective value. We use the dual value at CPLEX's root node as the upper bound for each instance.

Similarly to the set covering problem, experiments were performed with two types of objective function coefficients. The first, $c = \mathbf{1}$, yields the *combinatorial* set packing problem. For the second cost function, let nc_j again denote the number of constraints that include variable x_j, $j = 1,\ldots,n$. We chose the objective coefficient of variable x_j uniformly at random from the range $[0.75nc_j, 1.25nc_j]$. As a result, variables that participate in more constraints have a higher cost, thereby yielding harder set packing problems since this is a maximization problem. This cost vector yields the *weighted* set packing problem.

The results for the first class of instances are presented in Fig. 5.7. For all tested instances, the solution obtained from the BDD restriction was at least as good as the IP solution for all cost functions. As the bandwidth increases, the gap also increases for both techniques, as the upper bound obtained from CPLEX's root node deteriorates for larger bandwidths. However, the BDD gap does not increase as much as the IP gap, which is especially noticeable for the weighted case. We note that the difference in times between the BDD and IP restrictions are negligible and lie below one second.

The results for the second class of instances are presented in Fig. 5.8. For all instances tested, the BDD bound was at least as good as the bound obtained with IP, though the solution quality from restricted BDDs was particularly superior for the weighted case. Intuitively, since A is sparser, fewer BDD node states are possible in each layer, implying that less information is lost by restricting the BDD width.



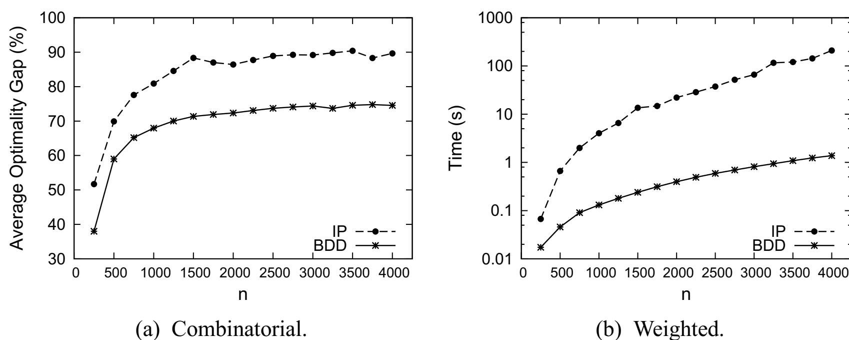
(a) Combinatorial. (b) Weighted.

Fig. 5.8 Average optimality gaps for combinatorial and weighted set packing instances with $n = 500$, varying k, and $b_w = 1.6k$.

Finally, we observe that times were also comparable for both IP and BDD cases, all below one second.

Next, we proceed analogously to the set covering case and compare solution quality and time as the number of variables n increases (Fig. 5.9). As before, we generate 30 random instances per configuration, with $n \in \{250, 500, 750, \ldots, 4000\}$, $k = 75$, and $b_w = 2.2k = 165$. Only weighted set packing instances are considered.

The average optimality gap and solution times are presented in Figs. 5.9(a) and 5.9(b), respectively. Similar to the set covering case, we observe that the BDD restrictions outperform the IP heuristics with respect to both gap and time for this particular configuration. The difference in gaps between restricted BDDs and IP remains approximately the same as n increases, while the time to generate restricted BDDs is orders of magnitude less than the IP times for the largest values of n tested.



(a) Combinatorial. (b) Weighted.

Fig. 5.9 Average optimality gaps and times for weighted set packing instances with varying n, $k = 75$, and $b_w = 2.2k = 165$. The y axis in the time plot is on logarithmic scale.

Chapter 6
Branch-and-Bound Based on Decision Diagrams

Abstract This chapter proposes an alternative branch-and-bound method in which decision diagrams take over the functions of the traditional relaxations and heuristics used in general-purpose optimization techniques. In particular, we show an enumeration scheme that branches on the *nodes* of a relaxed decision diagram, as opposed to variable-value assignments as in traditional branch-and-bound. We provide a computational study of our method on three classical combinatorial optimization problems, and compare our solution technology with mixed-integer linear programming. Finally, we conclude by showing how the diagram-based branch-and-bound procedure is suitable for *parallelization*, and provide empirical evidence of almost linear speedups on the maximum independent set problem.

6.1 Introduction

Some of the most effective methods for discrete optimization are branch-and-bound algorithms applied to an integer programming formulation of the problem. Linear programming (LP) relaxation plays a central role in these methods, primarily by providing bounds and feasible solutions as well as guidance for branching.

As we analyzed in Chapters 4 and 5, limited-size decision diagrams (DDs) can be used to provide useful relaxations and restrictions of the feasible set of an optimization problem in the form of relaxed and restricted DDs, respectively. We will use them in a novel *branch-and-bound* scheme that operates within a DD relaxation of the problem. Rather than branch on values of a variable, the scheme branches on a suitably chosen subset of nodes in the relaxed DD. Each node gives

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_6

rise to a subproblem for which a relaxed DD can be created, and so on recursively. This sort of branching implicitly enumerates sets of partial solutions, rather than values of one variable. It also takes advantage of information about the search space that is encoded in the structure of the relaxed DD. The branching nodes are selected on the basis of that structure, rather than on the basis of fractional variables, pseudo-costs, and other information obtained from an LP solution.

Because our DD-based solver is proposed as a general-purpose method, it is appropriate to compare it with another general-purpose solver. Integer programming is widely viewed as the most highly developed technology for general discrete optimization, and we therefore compare DD-based optimization with a leading commercial IP solver in Section 6.5. We find that, although IP solvers have improved by orders of magnitude since their introduction, our rudimentary DD-based solver is competitive with or superior to the IP state of the art on the tested problem instances.

Finally, we will show that the proposed branch-and-bound method can be easily *parallelized* by distributing the DD node processing (i.e., the construction of relaxed and restricted DDs) across multiple computers. This yields a low-communication parallel algorithm that is suitable for large clusters with hundreds or thousands of computers. We will also compare the parallel version of the branch-and-bound algorithm with IP, since it is a general-purpose solver with parallelization options, and show that our parallel method achieves almost linear speedups.

6.2 Sequential Branch-and-Bound

We now present our sequential DD-based branch-and-bound algorithm. We first define the notion of exact and relaxed nodes and indicate how they can be identified. Then, given a relaxed DD, we describe a technique that partitions the search space so that relaxed/restricted DDs can be used to bound the objective function for each subproblem. Finally, we present the branch-and-bound algorithm. For simplification, we focus on binary decision diagrams (BDDs), but the concepts presented here can be easily extended to multivalued decision diagrams.

For a given BDD B and nodes $u, u' \in B$ with $L(u) < L(u')$, we let $B_{uu'}$ be the BDD induced by the nodes that lie on directed paths from u to u' (with the same arc domains and arc cost as in B). In particular, $B_{rt} = B$.

6.3 Exact Cutsets

The branch-and-bound algorithm is based on enumerating subproblems defined by nodes in an exact cutset. To develop this idea, let $\overline{B}$ be a relaxed BDD created by Algorithm 1 using a valid DP model of the binary optimization problem $\mathscr{P}$. We say that a node $\overline{u}$ in $\overline{B}$ is *exact* if all r–$\overline{u}$ paths in $\overline{B}$ lead to the same state s^j. A *cutset* of $\overline{B}$ is a subset S of nodes of $\overline{B}$ such that any r–t path of $\overline{B}$ contains at least one node in S. We call a cutset *exact* if all nodes in S are exact.

As an illustration, Fig. 6.1(a) duplicates the relaxed BDD $\overline{B}$ from Fig. 3.4 and labels the nodes as *exact* (E) or *relaxed* (R). Node $\overline{u}_4$ in $\overline{B}$ is an exact node because all incoming paths (there is only one) lead to the same state $\{4,5\}$. Node $\overline{u}_3$ is relaxed because the two incoming paths represent partial solutions $(x_1,x_2) = (0,0)$ and $(0,1)$ that lead to different states, namely $\{3,4,5\}$ and $\{5\}$, respectively. Nodes $\overline{u}_1$ and $\overline{u}_4$ form one possible exact cutset of $\overline{B}$.

We now show that an exact cutset provides an exhaustive enumeration of subproblems. If B is an exact BDD for the binary optimization problem $\mathscr{P}$, let $v^*(B_{uu'})$ be the length of a longest u–u' path in $B_{uu'}$. For a node u in B, we define $\mathscr{P}|_u$ to be the restriction of $\mathscr{P}$ whose feasible solutions correspond to r–t paths of B that contain u. Recall that $z^*(\mathscr{P})$ is the optimal value of $\mathscr{P}$.

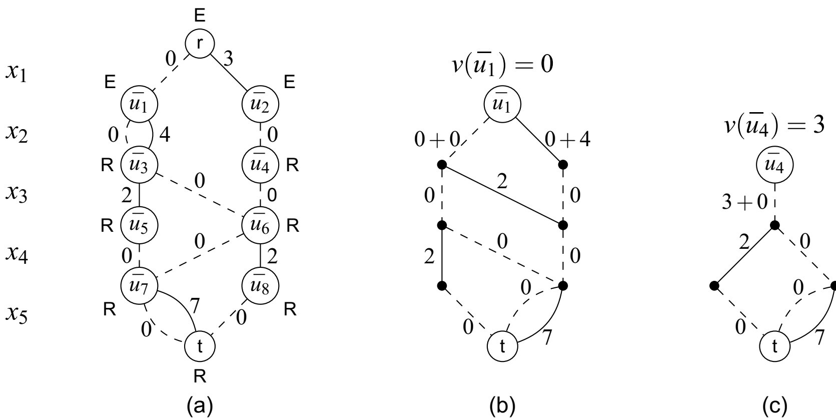


Fig. 6.1 (a) Relaxed BDD for the MISP on the graph Fig. 3.3 with nodes labeled as exact (E) or relaxed (R); (b) exact BDD for subproblem corresponding to $\overline{u}_1$; (c) exact BDD for subproblem corresponding to $\overline{u}_4$.

Lemma 6.1. *If B is an exact BDD for $\mathscr{P}$, then for any node u in B,*

$$v^*(B_{ru}) + v^*(B_{ut}) = z^*(\mathscr{P}|_u).$$

Proof. $z^*(\mathscr{P}|_u)$ is the length of a longest r–t path of B that contains u, and any such path has length $v^*(B_{ru}) + v^*(B_{ut})$. □

Theorem 6.1. *Let $\bar{B}$ be a relaxed BDD created by Algorithm 3 using a valid DP model of the binary optimization problem $\mathscr{P}$, and let S be an exact cutset of $\bar{B}$. Then*

$$z^*(\mathscr{P}) = \max_{u \in S} \{z^*(\mathscr{P}|_u)\}.$$

Proof. Let B be the exact BDD for $\mathscr{P}$ created using the same DP model. Because each node $\bar{u} \in S$ is exact, it has a corresponding node u in B (i.e., a node associated with the same state), and S is a cutset of B. Thus

$$z^*(\mathscr{P}) = \max_{u \in S} \{v^*(B_{ru}) + v^*(B_{ut})\} = \max_{u \in S} \{\mathscr{P}|_u\},$$

where the second equation is due to Lemma 6.1. □

6.4 Enumeration of Subproblems

We solve a binary optimization problem $\mathscr{P}$ by a branching procedure in which we enumerate a set of subproblems $\mathscr{P}|_u$ each time we branch, where u ranges over the nodes in an exact cutset of the current relaxed BDD. We build a relaxed BDD and a restricted BDD for each subproblem to obtain upper and lower bounds, respectively.

Suppose u is one of the nodes on which we branch. Because u is an exact node, we have already constructed an exact BDD B_{ru} down to u, and we know the length $v^*(u) = v^*(B_{ru})$ of a longest path in B_{ru}. We can obtain an upper bound on $z^*(\mathscr{P}|_u)$ by computing a longest path length $v^*(B_{ut})$ in a relaxed BDD $\bar{B}_{ut}$ with root value $v^*(u)$. To build the relaxation $\bar{B}_{ut}$, we start the execution of Algorithm 1 with $j = L(u)$ and root node u, where the root value is $v_r = v^*(u)$. We can obtain a lower bound on $z^*(\mathscr{P}|_u)$ in a similar fashion, except that we use a restricted rather than a relaxed BDD.

The branch-and-bound algorithm is presented in Algorithm 6. We begin with a set $Q = \{r\}$ of open nodes consisting of the initial state r of the DP model. Then,

Algorithm 6 Branch-and-Bound Algorithm

```
1: initialize Q = {r}, where r is the initial DP state
2: let z_opt = −∞, v*(r) = 0
3: while Q ≠ ∅ do
4:    u ← select_node(Q), Q ← Q \ {u}
5:    create restricted BDD B'_ut using Algorithm 1 with root u and v_r = v*(u)
6:    if v*(B_ut) > z_opt then
7:       z_opt ← v*(B')
8:    if B'_ut is not exact then
9:       create relaxed BDD B̄_ut using Algorithm 1 with root u and v_r = v*(u)
10:      if v*(B̄_ut) > z_opt then
11:         let S be an exact cutset of B̄_ut
12:         for all u' ∈ S do
13:            let v*(u') = v*(u) + v*(B̄_uu'), add u' to Q
14: return z_opt
```

while open nodes remain, we select a node u from Q. We first obtain a lower bound on $z^*(\mathscr{P}|_u)$ by creating a restricted BDD B'_{ut} as described above, and we update the incumbent solution z_{opt}. If B'_{ut} is exact (i.e., $|L_j|$ never exceeds W in Algorithm 5), there is no need for further branching at node u. This is analogous to obtaining an integer solution in traditional branch-and-bound. Otherwise we obtain an upper bound on $z^*(\mathscr{P}|_u)$ by building a relaxed BDD $\overline{B}_{ut}$ as described above. If we cannot prune the search using this bound, we identify an exact cutset S of $\overline{B}_{ut}$ and add the nodes in S to Q. Because S is exact, for each $u' \in S$ we know that $v^*(u') = v^*(u) + v^*(\overline{B}_{uu'})$. The search terminates when Q is empty, at which point the incumbent solution is optimal by Theorem 6.1.

As an example, consider again the relaxed BDD $\overline{B}$ in Fig. 6.1(a). The longest path length in this graph is $v^*(\overline{B}) = 13$, an upper bound on the optimal value. Suppose that we initially branch on the exact cutset $\{\overline{u}_1, \overline{u}_4\}$, for which we have $v(\overline{u}_1) = 0$ and $v(\overline{u}_4) = 3$. We wish to generate restricted and relaxed BDDs of maximum width 2 for the subproblems. Figure 6.1(b) shows a restricted BDD $\overline{B}_{\bar{u}_1 t}$ for the subproblem at $\bar{u}_1$, and Fig. 6.1(c) shows a restricted BDD $\overline{B}_{\bar{u}_4 t}$ for the other subproblem. As it happens, both BDDs are exact, and so no further branching is necessary. The two BDDs yield bounds $v^*(\overline{B}_{\bar{u}_1 t}) = 11$ and $v^*(\overline{B}_{\bar{u}_4 t}) = 10$, respectively, and so the optimal value is 11.

6.4.1 Exact Cutset Selection

Given a relaxed BDD, there are many exact cutsets. Here we present three such cutsets and experimentally evaluate them in Section 6.5.

- *Traditional branching (TB)*. Branching normally occurs by selecting some variable x_j and branching on $x_j = 0/1$. Using the exact cutset $S = L_2$ has the same effect. Traditional branching therefore uses the shallowest possible exact cutset for some variable ordering.
- *Last exact layer (LEL)*. For a relaxed BDD $\overline{B}$, define the *last exact layer* of $\overline{B}$ to be the set of nodes $\mathrm{LEL}(\overline{B}) = L_{j'}$, where j' is the maximum value of j for which each node in L_j is exact. In the relaxed BDD $\overline{B}$ of Fig. 6.1(a), $\mathrm{LEL}(\overline{B}) = \{\overline{u}_1, \overline{u}_2\}$.
- *Frontier cutset (FC)*. For a relaxed BDD $\overline{B}$, define the *frontier cutset* of B to be the set of nodes

$$\mathrm{FC}(\overline{B}) = \{u \text{ in } \overline{B} \mid u \text{ is exact and } b_0(u) \text{ or } b_1(u) \text{ is relaxed}\}.$$

In the example of Fig. 6.1(a), $\mathrm{FC}(\overline{B}) = \{\overline{u}_1, \overline{u}_4\}$. A frontier cutset is an exact cutset due to the following lemma:

Lemma 6.2. *If $\overline{B}$ is a relaxed BDD that is not exact, then* $\mathrm{FC}(\overline{B})$ *is an exact cutset.*

Proof. By the definition of a frontier cutset, each node in the cutset is exact. We need only show that each solution $x \in \mathrm{Sol}(\overline{B})$ contains some node in $\mathrm{FC}(\overline{B})$. But the path p corresponding to x ends at t, which is relaxed because $\overline{B}$ is not exact. Since the root r is exact, there must be a first relaxed node u in p. The node immediately preceding this node in p is in $\mathrm{FC}(\overline{B})$, as desired. □

6.5 Computational Study

Since we propose BDD-based branch-and-bound as a general discrete optimization method, it is appropriate to measure it against an existing general-purpose method. We compared BDDs with a state-of-the-art IP solver, inasmuch as IP is generally viewed as a highly developed general-purpose solution technology for discrete optimization.

Like IP, a BDD-based method requires several implementation decisions, chief among which are the following:

- *Maximum width*: Wider relaxed BDDs provide tighter bounds but require more time to build. For each subproblem in the branch-and-bound procedure, we set the maximum width W equal to the number of variables whose value has not yet been fixed.
- *Node selection for merger*: The selection of the subset M of nodes to merge during the construction of a relaxed BDD (line 4 of Algorithm 1) likewise affects the quality of the bound, as discussed in Chapters 4 and 5. We use the following heuristic: After constructing each layer L_j of the relaxed BDD, we rank the nodes in L_j according to a rank function $\text{rank}(u)$ that is specified in the DP model with the state merging operator $\oplus$. We then let M contain the lowest-ranked $|L_j| - W$ nodes in L_j.
- *Variable ordering*: Much as branching order has a significant impact on IP performance, the variable ordering chosen for the layers of the BDD can affect branching efficiency and the tightness of the BDD relaxation. We describe below the variable ordering heuristics we used for the three problem classes.
- *Search node selection*: We must also specify the next node in the set Q of open nodes to be selected during branch-and-bound (Algorithm 6). We select the node u with the minimum value $v^*(u)$.

The tests were run on an Intel Xeon E5345 with 8 GB RAM. The BDD-based algorithm was implemented in C++. The commercial IP solver ILOG CPLEX 12.4 was used for comparison. Default settings, including presolve, were used for CPLEX unless otherwise noted. No presolve routines were used for the BDD-based method.

6.5.1 Results for the MISP

We first specify the key elements of the algorithm that we used for the MISP. Node selection for merger is based on the rank function $\text{rank}(u) = v^*(u)$. For variable ordering, we considered the heuristic **minState** first described in Section 4.6.2: after selecting the first $j-1$ variables and forming layer L_j, we choose vertex j as the vertex that belongs to the fewest number of states in L_j. Finally, we used FC cutsets for all MISP tests.

For a graph $G = (V,E)$, a standard IP model for the MISP is

$$\max\left\{ \sum_{i\in V} x_i \;\middle|\; x_i + x_j \leq 1, \text{ all } (i,j) \in E;\; x_i \in \{0,1\}, \text{ all } i \in V \right\}. \tag{6.1}$$

A tighter linear relaxation can be obtained by precomputing a clique cover $\mathscr{C}$ of G and using the model

$$\max\left\{ \sum_{i\in S} x_i \;\middle|\; x_i \leq 1, \text{ all } S \in \mathscr{C};\; x_i \in \{0,1\}, \text{ all } i \in V \right\}. \tag{6.2}$$

We refer to this as the *tight* MISP formulation. The clique cover $\mathscr{C}$ is computed using a greedy procedure: Starting with $\mathscr{C} = \emptyset$, let clique S consist of a single vertex v with the highest positive degree in G. Add to S the vertex with highest degree in $G \setminus S$ that is adjacent to all vertices in S, and repeat until no more additions are possible. At this point, add S to $\mathscr{C}$, remove from G all the edges of the clique induced by S, update the vertex degrees, and repeat the overall procedure until G has no more edges.

We begin by reporting results on randomly generated graphs. We generated random graphs with $n \in \{250, 500, \ldots, 1,750\}$ and density $p \in \{0.1, 0.2, \ldots, 1\}$ (10 graphs per n,p configuration) according to the Erdös–Rényi model $G(n,p)$ (where each edge appears independently with probability p).

Figure 6.2 depicts the results. The solid lines represent the average percent gap for the BDD-based technique after 1,800 seconds, one line per value of n, and the dashed lines depict the same statistics for the integer programming solver using the tighter, clique model, only. It is clear that the BDD-based algorithm outperforms CPLEX on dense graphs, solving all instances tested with density 80% or higher, and solving almost all instances, except for the largest, with density equal to 70%, whereas the integer programming solver could not close any but the smallest instances (with $n = 250$) at these densities.

CPLEX outperformed the BDD technique for the sparsest graphs (with $p = 10$), but only for the small values of n. As n grows, we see that the BDD-based algorithm starts to outperform CPLEX, even on the sparsest graphs, and that the degree to which the ending percent gaps increase as n grows is more substantial for CPLEX than it is for the BDD-based algorithm.

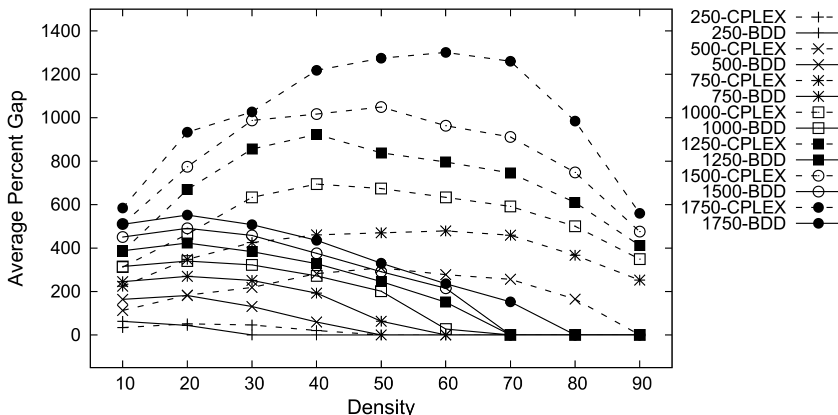


Fig. 6.2 Average percent gap on randomly generated MISP instances.

We also tested on the 87 instances of the maximum clique problem in the well-known DIMACS benchmark (http://cs.hbg.psu.edu/txn131/clique.html). The MISP is equivalent to the maximum clique problem on the complement of the graph.

Figure 6.3 shows a time profile comparing BDD-based optimization with CPLEX performance for the standard and tight IP formulations. The BDD-based algorithm is superior to the standard IP formulation but solved four fewer instances than the tight IP formulation after 30 minutes. However, fewer than half the instances were solved by any method. The relative gap (upper bound divided by lower bound) for the remaining instances therefore becomes an important factor. A comparison of the

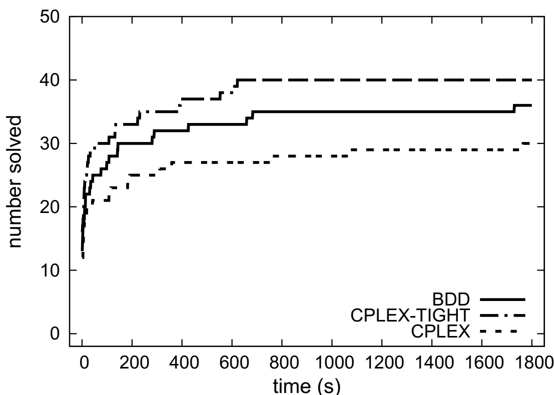


Fig. 6.3 Results on 87 MISP instances for BDDs and CPLEX. Number of instances solved versus time for the tight IP model (top line), BDDs (middle), standard IP model (bottom).

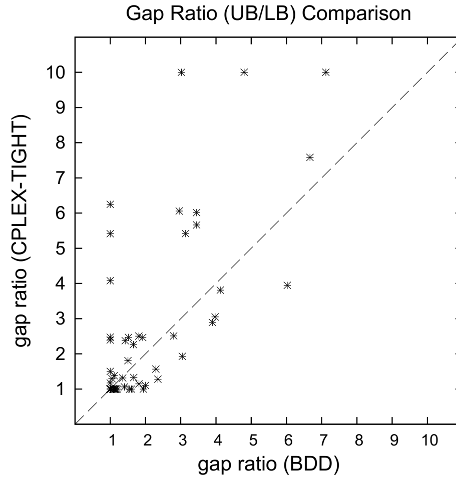


Fig. 6.4 Results on 87 MISP instances for BDDs and CPLEX. End gap comparison after 1800 seconds.

relative gap for BDDs and the tight IP model appears in Fig. 6.4, where the relative gap for CPLEX is shown as 10 when it found no feasible solution. Points above the diagonal are favorable to BDDs. It is evident that BDDs tend to provide significantly tighter bounds. There are several instances for which the CPLEX relative gap is twice the BDD gap, but no instances for which the reverse is true. In addition, CPLEX was unable to find a lower bound for three of the largest instances, while BDDs provided bounds for all instances.

6.5.2 Results for the MCP

We evaluated our approach on random instances for the MCP. For $n \in \{30,40,50\}$ and $p \in \{0.1,0.2,\ldots,1\}$, we again generated random graphs (10 per n,p configuration). The weights of the edges generated were drawn uniformly from $[-1,1]$.

We let the rank of a node $u \in L_j$ associated with state s^j be

$$\text{rank}(u) = v^*(u) + \sum_{\ell=j}^{n} \left| s^j_\ell \right|.$$

We order the variables x_j according to the sum of the lengths of the edges incident to vertex j. Variables with the largest sum are first in the ordering.

A traditional IP formulation of the MCP introduces a 0–1 variable y_{ij} for each edge $(i,j) \in E$ to indicate whether this edge crosses the cut. The formulation is

$$\min \left\{ \sum_{(i,j)\in E} w_{ij} y_{ij} \;\middle|\; \begin{cases} y_{ij} + y_{ik} + y_{jk} \le 2 \\ y_{ij} + y_{ik} \ge y_{jk} \end{cases} \right. \\ \left. \text{all } i,j,k \in \{1,\dots,n\};\; y_{ij} \in \{0,1\}, \text{all } (i,j) \in E \right\}.$$

We first consider instances with $n = 30$ vertices, all of which were solved by both BDDs and IP within 30 minutes. Figure 6.5 shows the average solution time for CPLEX and the BDD-based algorithm, using both LEL and FC cutsets for the latter. We tested CPLEX with and without presolve because presolve reduces the model size substantially. We find that BDDs with either type of cutset are substantially faster than CPLEX, even when CPLEX uses presolve. In fact, the LEL solution time for BDDs is scarcely distinguishable from zero in the plot. The advantage of BDDs is particularly great for denser instances.

Results for $n = 40$ vertices appear in Fig. 6.6. BDDs with LEL are consistently superior to CPLEX, solving more instances after 1 minute and after 30 minutes. In fact, BDD solved all but one of the instances within 30 minutes, while CPLEX with presolve left 17 unsolved.

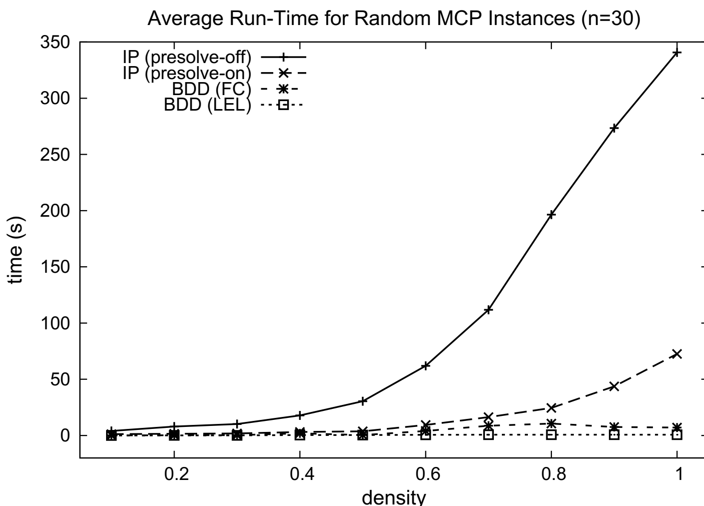


Fig. 6.5 Average solution time for MCP instances ($n = 30$ vertices) using BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve). Each point is the average of 10 random instances.

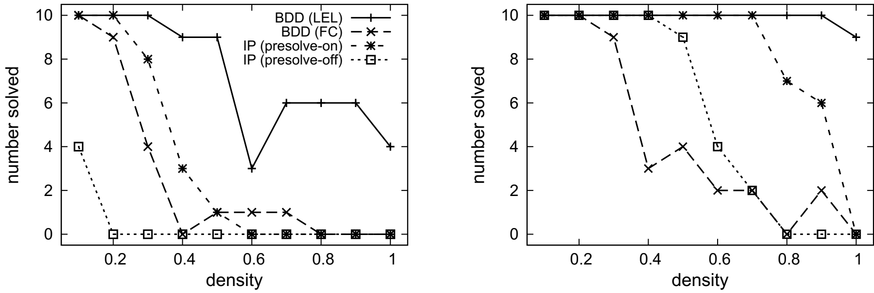


Fig. 6.6 Number of MCP instances with $n = 40$ vertices solved after 60 seconds (left) and 1800 seconds (right), versus graph density, using BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve). The legend is the same for the two plots.

Figure 6.7 (left) shows time profiles for 100 instances with $n = 50$ vertices. The profiles for CPLEX (with presolve) and BDDs (with LEL) are roughly competitive, with CPLEX marginally better for larger time periods. However, none of the methods could solve even a third of the instances, and so the gap for the remaining instances becomes important. Figure 6.7 (right) shows that the average percent gap (i.e., $100(\mathrm{UB}-\mathrm{LB})/\mathrm{LB}$) is much smaller for BDDs on denser instances, and comparable on sparser instances, again suggesting greater robustness for a BDD-based method relative to CPLEX. In view of the fact that CPLEX benefits enormously from presolve, it is conceivable that BDDs could likewise profit from a presolve routine.

We also tested the algorithm on the g-set, a classical benchmark set, created by the authors in [91], which has since been used extensively for computational testing on algorithms designed to solve the MCP. The 54 instances in the benchmark set are

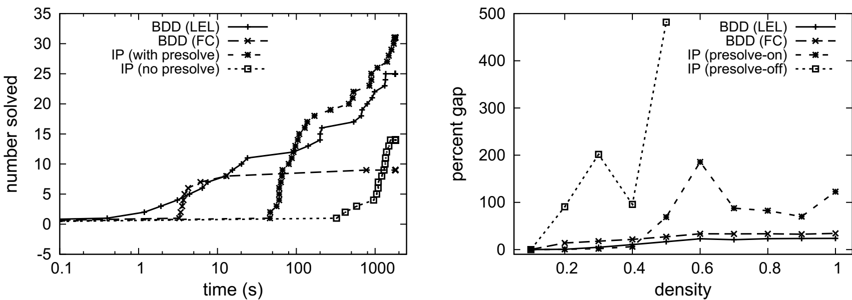


Fig. 6.7 Time profile (left) for 100 MCP instances with $n = 50$ vertices, comparing BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve). Percent gap (right) versus density after 1800 seconds, where each point is the average over 10 random instances.

large, each having at least 800 vertices. The results appear in Table 6.1 only for those instances for which the BDD-based algorithm was able to improve upon the best known integrality gaps. For the instances with 1% density or more, the integrality gap provided by the BDD-based algorithm is about an order of magnitude worse than the best known integrality gaps, but for these instances (which are among the sparsest), we are able to improve on the best known gaps through proving tighter relaxation bounds and identifying better solutions than have ever been found.

The first column provides the name of the instance. The instances are ordered by density, with the sparsest instances reported appearing at the top of the table. We then present the upper bound (UB) and lower bound (LB), after one hour of computation time, for the BDD-based algorithm, follow by the best known upper bound and lower bound that we could find in the literature. In the final columns, we record the previously best known percent gap and the new percent gap obtained from the BDD-based algorithm. Finally, we present the reduction in percent gap obtained.

For three instances (g32, g33, and g34), better solutions were identified by the BDD-based algorithm than have ever been found by any technique, with an improvement in objective function value of 12, 4, and 4, respectively. In addition, for four instances (g50, g33, g11, and g12) better upper bounds were proven than were previously known, reducing the best known upper bound by 89.18, 1, 60, and 5, respectively. For these instances, the reduction in the percent gap is shown in the last column. Most notably, for g50 and g11, the integrality gap was significantly tightened (82.44 and 95.24 percent reduction, respectively). As the density grows, however, the BDD-based algorithm is not able to compete with other state-of-the-art techniques, yielding substantially worse solutions and relaxation bounds than the best known values.

We note here that the BDD-based technique is a general branch-and-bound procedure, whose application to the MCP is only specialized through the DP model that is used to calculate states and determine transition costs. This general technique was able to improve upon best known solutions obtained by heuristics and exact techniques specifically designed to solve the MCP. And so, although the technique is unable to match the best known objective function bounds for all instances, identifying the best known solution via this general-purpose technique is an indication of the power of the algorithm.

Table 6.1 g-set computational results

Instance	BDDs		Previous best		Percent gap		
	UB	LB	UB	LB	Previous	BDDs	% reduction
g50	**5899**	5880	5988.18	5880	1.84	**0.32**	**82.44**
g32	1645	**1410**	1560	1398	11.59	**10.64**	**8.20**
g33	**1536**	**1380**	1537	1376	11.7	**11.30**	**3.39**
g34	1688	**1376**	1541	1372	12.32	**11.99**	**2.65**
g11	**567**	564	627	564	11.17	**0.53**	**95.24**
g12	**616**	556	621	556	11.69	**10.79**	**7.69**

6.5.3 Results for MAX-2SAT

For the MAX-2SAT problem, we created random instances with $n \in \{30, 40\}$ variables and density $d \in \{0.1, 0.2, \dots, 1\}$. We generated 10 instances for each pair (n, d), with each of the $4 \cdot \binom{n}{2}$ possible clauses selected with probability d and, if selected, assigned a weight drawn uniformly from $[1, 10]$.

We used the same rank function as for the MCP, and we ordered the variables in ascending order according to the total weight of the clauses in which the variables appear.

We formulated the IP using a standard model. Let clause i contain variables $x_{j(i)}$ and $x_{k(i)}$. Let x^i_j be x_j if x_j is posited in clause i, and $1 - x_j$ if x_j is negated. Let δ_i be a 0–1 variable that will be forced to 0 if clause i is unsatisfied. Then if there are m clauses and w_i is the weight of clause i, the IP model is

$$\max \left\{ \sum_{i=1}^{m} w_i \delta_i \;\middle|\; x^i_{j(i)} + x^i_{k(i)} + (1 - \delta_i) \geq 1, \text{all } i;\ x_j, \delta_i \in \{0, 1\}, \text{ all } i, j \right\}.$$

Figure 6.8 shows the time profiles for the two size classes. BDDs with LEL are clearly superior to CPLEX for $n = 30$. When $n = 40$, BDDs prevail over CPLEX as the available solving time grows. In fact, BDDs solve all but 2 of the instances within 30 minutes, while CPLEX leaves 17 unsolved using no presolve, and 22 unsolved using presolve.

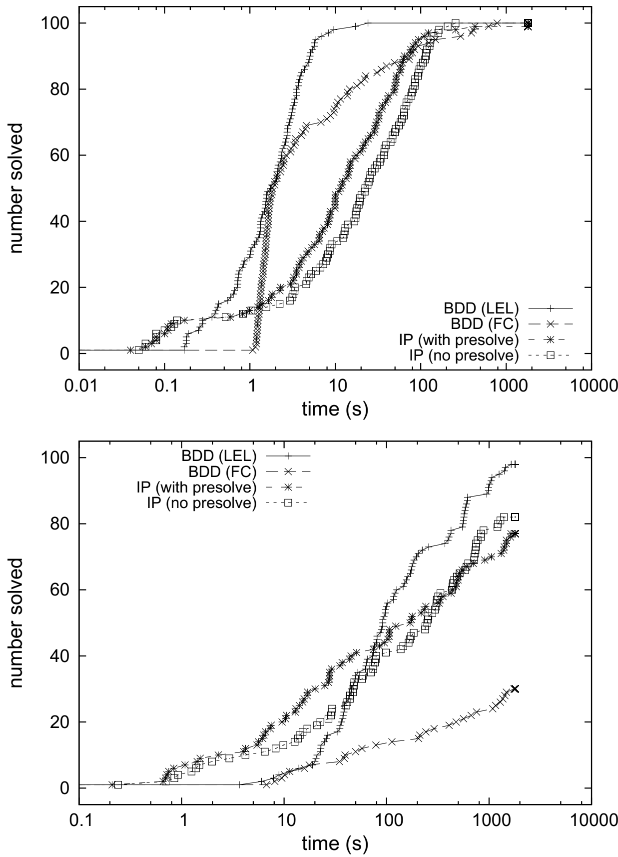


Fig. 6.8 Time profile for 100 MAX-2SAT instances with $n = 30$ variables (top) and $n = 40$ variables (bottom), comparing BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve).

6.6 Parallel Branch-and-Bound

In recent years, hardware design has increasingly focused on multicore systems and parallelized computing. In order to take advantage of these systems, it is crucial that solution methods for combinatorial optimization be effectively parallelized and built to run not only on one machine but also on a large cluster.

Different combinatorial search methods have been developed for specific problem classes, including mixed-integer programming (MIP), Boolean satisfiability (SAT), and constraint programming (CP). These methods represent (implicitly or explicitly) a complete enumeration of the solution space, usually in the form of a

branching tree where the branches out of each node reflect variable assignments. The recursive nature of branching trees suggests that combinatorial search methods are amenable to efficient parallelization, since we may distribute subtrees to different compute cores spread across multiple machines of a compute cluster.

Yet, in practice this task has proved to be very challenging. For example, Gurobi, one of the leading commercial MIP solvers, achieves an average speedup factor of 1.7 on 5 machines (and only 1.8 on 25 machines) when compared with using only 1 machine [79]. Furthermore, during the 2011 SAT Competition, the best parallel SAT solvers obtained an average speedup factor of about 3 on 32 cores, which was achieved by employing an algorithm portfolio rather than a parallelized search [102]. The winner of the parallel category of the 2013 SAT Competition also achieved a speedup of only about 3 on 32 cores.

Constraint programming search appears to be more suitable for parallelization than search for MIP or SAT. Different strategies, including a recursive application of search goals [125], work stealing [48], problem decomposition [135], and a dedicated parallel scheme based on limited discrepancy search [118] all exhibit good speedups (sometimes near-linear). This is specially true in scenarios involving infeasible instances or where evaluating the search tree leaves is costlier than evaluating internal nodes. Nonetheless, recent developments in CP and SAT have moved towards more constraint *learning* during search (such as *lazy clause* generation [120]) for which efficient parallelization becomes increasingly more difficult.

Our goal in this section is to investigate whether branch-and-bound based on decision diagrams can be effectively parallelized. The key observation is that relaxed decision diagrams can be used to partition the search space, since for a given layer in the diagram each path from the root to the terminal passes through a node in that layer. We can therefore *branch on nodes in the decision diagram* instead of branching on variable–value pairs, as is done in conventional search methods. Each of the subproblems induced by a node in the diagram is processed recursively, and the process continues until all nodes have been solved by an exact decision diagram or pruned due to reasoning based on bounds on the objective function.

When designing parallel algorithms geared towards dozens or perhaps hundreds of workers operating in parallel, the two major challenges are (1) balancing the workload across the workers, and (2) limiting the communication cost between workers. In the context of combinatorial search and optimization, most of the current methods are based on either parallelizing the traditional tree search or using portfolio techniques that make each worker operate on the entire problem.

The former approach makes load balancing difficult as the computational cost of solving similarly sized subproblems can be orders of magnitude different. The latter approach typically relies on extensive communication in order to avoid duplication of effort across workers.

In contrast, using decision diagrams as a starting point for parallelization offers several notable advantages. For instance, the associated branch-and-bound method applies relaxed and restricted diagrams that are obtained by limiting the size of the diagrams to a certain maximum value. The size can be controlled, for example, simply by limiting the maximum width of the diagrams. As the computation time for a (sub)problem is roughly proportional to the size of the diagram, by controlling the size we are able to control the computation time. In combination with the recursive nature of the framework, this makes it easier to obtain a balanced workload. Further, the communication between workers can be limited in a natural way by using both global and local pools of currently open subproblems and employing pruning based on shared bounds. Upon processing a subproblem, each worker generates several new ones. Instead of communicating all of these back to the global pool, the worker keeps several of them to itself and continues to process them. In addition, whenever a worker finds a new feasible solution, the corresponding bound is communicated immediately to the global pool as well as to other workers, enabling them to prune subproblems that cannot provide a better solution. This helps avoid unnecessary computational effort, especially in the presence of local pools.

Our scheme is implemented in X10 [42, 139, 158], which is a modern programming language designed specifically for building applications for multicore and clustered systems. For example, [34] recently introduced SatX10 as an efficient and generic framework for parallel SAT solving using X10. We refer to our proposed framework for parallel decision diagrams as DDX10. The use of X10 allows us to program parallelization and communication constructs using a high-level, type-checked language, leaving the details of an efficient backend implementation for a variety of systems and communication hardware to the language compiler and run-time. Furthermore, X10 also provides a convenient parallel execution framework, allowing a single compiled executable to run as easily on one core as on a cluster of networked machines.

Our main contributions are as follows: First, we describe, at a conceptual level, a scheme for parallelization of a sequential branch-and-bound search based on approximate decision diagrams and discuss how this can be efficiently implemented in the X10 framework. Second, we provide an empirical evaluation on the maximum

independent set problem, showing the potential of the proposed method. Third, we compare the performance of DDX10 with a state-of-the-art parallel MIP solver. Experimental results indicate that DDX10 can obtain much better speedups than parallel MIP, especially when more workers are available. The results also demonstrate that the parallelization scheme provides near-linear speedups up to 256 cores, even in a distributed setting where the cores are split across multiple machines.

The limited amount of information required for each BDD node makes the branch-and-bound algorithm naturally suitable for parallel processing. Once an exact cut C is computed for a relaxed BDD, the nodes $u \in C$ are independent and can each be processed in parallel. The information required to process a node $u \in C$ is its corresponding state, which is bounded by the number of vertices of G, $|V|$. After processing a node u, only the lower bound $v^*(G[E(u)])$ is needed to compute the optimal value.

6.6.1 A Centralized Parallelization Scheme

There are many possible parallel strategies that can exploit this natural characteristic of the branch-and-bound algorithm for approximate decision diagrams. We propose here a *centralized strategy*. Specifically, a *master process* keeps a pool of BDD nodes to be processed, first initialized with a single node associated with the root state V. The master distributes the BDD nodes to a set of *workers*. Each worker receives a number of nodes, processes them by creating the corresponding relaxed and restricted BDDs, and either sends back to the master new nodes to explore (from an exact cut of their relaxed BDD) or sends to the master as well as all workers an improved lower bound from a restricted BDD.

The workers also send the upper bound obtained from the relaxed BDD from which the nodes were extracted, which is then used by the master for potentially pruning the nodes according to the current best lower bound at the time these nodes are brought out from the global pool to be processed.

Even though conceptually simple, our centralized parallelization strategy involves communication between all workers and many choices that have a significant impact on performance. After discussing the challenge of effective parallelization, we explore some of these choices in the rest of this section.

6.6.2 The Challenge of Effective Parallelization

Clearly, a BDD constructed in parallel as described above can be very different in structure and overall size from a BDD constructed sequentially for the same problem instance. As a simple example, consider two nodes u_1 and u_2 in the exact cut C. By processing u_1 first, one could potentially improve the lower bound so much that u_2 can be pruned right away in the sequential case. In the parallel setting, however, while worker 1 processes u_1, worker 2 will already be wasting search effort on u_2, not knowing that u_2 could simply be pruned if it waited for worker 1 to finish processing u_1.

In general, the order in which nodes are processed in the approximate BDD plays a key role in perfomance. The information passed on by nodes processed earlier can substantially alter the direction of search later. This is very clear in the context of combinatorial search for SAT, where dynamic variable activities and clauses learned from conflicts dramatically alter the behavior of subsequent search. Similarly, bounds in MIP and impacts in CP influence subsequent search.

Issues of this nature pose a challenge to effective parallelization of anything but brute-force combinatorial search oblivious to the order in which the search space is explored. Such a search is, of course, trivial to parallelize. For most search methods of interest, however, a parallelization strategy that delicately balances independence of workers with timely sharing of information is often the key to success. As our experiments will demonstrate, our implementation, DDX10, achieves this balance to a large extent on both random and structured instances of the independent set problem. In particular, the overall size of parallel BDDs is not much larger than that of the corresponding sequential BDDs. In the remainder of this section, we discuss the various aspects of DDX10 that contribute to this desirable behavior.

6.6.3 Global and Local Pools

We refer to the pool of nodes kept by the master as the *global pool*. Each node in the global pool has two pieces of information: a state, which is necessary to build the relaxed and restricted BDDs, and the longest path value in the relaxed BDD that created that node, from the root to the node. All nodes sent to the master are first stored in the global pool and then redistributed to the workers. Nodes with an upper

bound that is no more than the best found lower bound at the time are pruned from the pool, as these can never provide a solution better than one already found.

In order to select which nodes to send to workers first, the global pool is implemented here using a data structure that mixes a priority queue and a stack. Initially, the global pool gives priority to nodes that have a larger upper bound, which intuitively are nodes with higher potential to yield better solutions. However, this search strategy simulates a best-first search and may result in an exponential number of nodes in the global queue that still need to be explored. To remedy this, the global pool switches to a *last-in, first-out* node selection strategy when its size reaches a particular value (denoted *maxPQueueLength*), adjusted according to the available memory on the machine where the master runs. This strategy resembles a stack-based depth-first search and limits the total amount of memory necessary to perform search.

Besides the global pool, workers also keep a *local pool* of nodes. The subproblems represented by the nodes are usually small, making it advantageous for workers to keep their own pool so as to reduce the overall communication to the master. The local pool is represented by a priority queue, selecting nodes with a larger upper bound first. After a relaxed BDD is created, a certain fraction of the nodes (with preference for those with a larger upper bound) in the exact cut are sent to the master, while the remaining fraction (denoted *fracToKeep*) of nodes are added to the local pool. The local pool size is also limited; when the pool reaches this maximum size (denoted *maxLocalPoolSize*), we stop adding more nodes to the local queue and start sending any newly created nodes directly to the master. When a worker's local pool becomes empty, it notifies the master that it is ready to receive new nodes.

6.6.4 Load Balancing

The global queue starts off with a single node corresponding to the root state V. The root assigned to an arbitrary worker, which then applies a cut to produce more states and sends a fraction of them, as discussed above, back to the global queue. The size of the global pool thus starts to grow rapidly, and one must choose how many nodes to send subsequently to other workers. Sending one node (the one with the highest priority) to a worker at a time would mimic the sequential case most closely. However, it would also result in the most number of communications between the master and the workers, which often results in a prohibitively large system overhead.

On the other hand, sending too many nodes at once to a single worker runs the risk of starvation, i.e., the global queue becoming empty and other workers sitting idle waiting to receive new work.

Based on experimentation with representative instances, we propose the following parameterized scheme to dynamically decide how many nodes the master should send to a worker at any time. Here, we use the notation $[x]_\ell^u$ as a shorthand for $\min\{u, \max\{\ell, x\}\}$, that is, x capped to lie in the interval $[\ell, u]$.

$$nNodesToSend_{c,\bar{c},c^*}(s,q,w) = \left[\min\left\{\bar{c}s, c^*\frac{q}{w}\right\}\right]_c^\infty, \tag{6.3}$$

where s is a decaying running average of the number of nodes added to the global pool by workers after processing a node,[1] q is the current size of the global pool, w is the number of workers, and $c, \bar{c}$, and c^* are parametrization constants.

The intuition behind this choice is as follows: c is a flat lower limit (a relatively small number) on how many nodes are sent at a time irrespective of other factors. The inner minimum expression upper bounds the number of nodes to send to be no more than both (a constant times) the number of nodes the worker is in turn expected to return to the global queue upon processing each node and (a constant times) an even division of all current nodes in the queue into the number of workers. The first influences how fast the global queue grows, while the second relates to fairness among workers and the possibility of starvation. Larger values of $c, \bar{c}$, and c^* reduce the number of times communication occurs between the master and workers, at the expense of moving further away from mimicking the sequential case.

Load balancing also involves appropriately setting the *fracToKeep* value discussed earlier. We use the following scheme, parameterized by d and d^*:

$$fracToKeep_{d,d^*}(t) = [t/d^*]_d^1, \tag{6.4}$$

where t is the number of states received by the worker. In other words, the fraction of nodes to keep for the local queue is $1/d^*$ times the number of states received by the worker, capped to lie in the range $[d, 1]$.

[1] When a cut C is applied upon processing a node, the value of s is updated as $s_{\text{new}} = rs_{\text{old}} + (1-r)|C|$, with $r = 0.5$ in the current implementation.

6.6.5 DDX10: Implementing Parallelization Using X10

As mentioned earlier, X10 is a high-level parallel programming and execution framework. It supports parallelism natively, and applications built with it can be compiled to run on various operating systems and communication hardware.

Similar to SatX10 [34], we capitalize on the fact that X10 can incorporate existing libraries written in C++ or Java. We start off with the sequential version of the BDD code base for MISP used in Section 6.5 and integrate it in X10, using the C++ backend. The integration involves adding hooks to the BDD class so that (a) the master can communicate a set of starting nodes to build approximate BDDs, (b) each worker can communicate nodes (and corresponding upper bounds) of an exact cut back to the master, and (c) each worker can send updated lower bounds immediately to all other workers and the master so as to enable pruning.

The global pool for the master is implemented natively in X10 using a simple combination of a priority queue and a stack. The DDX10 framework itself (consisting mainly of the main DDSolver class in DDX10.x10 and the pool in StatePool.x10) is generic and not tied to MISP in any way. It can, in principle, work with any maximization or minimization problem for which states for a BDD (or even an MDD) can be appropriately defined.

6.6.6 Computational Study

The MISP problem can be formulated and solved using several existing general-purpose discrete optimization techniques. A MIP formulation is considered to be very effective and has been used previously to evaluate the sequential BDD approach in Section 6.5. Given the availability of parallel MIP solvers as a comparison point, we present two sets of experiments on the MISP problem: (1) we compare DDX10 with a MIP formulation solved using IBM ILOG CPLEX 12.5.1 on up to 32 cores, and (2) we show how DDX10 scales when going beyond 32 cores and employing up to 256 cores distributed across a cluster. We borrow the MIP encoding from Section 6.5 and employ the built-in parallel branch-and-bound MIP search mechanism of CPLEX. The comparison with CPLEX is limited to 32 cores because this is the largest number of cores we have available on a single machine (note that CPLEX 12.5.1 does not support distributed execution). Since the current version of DDX10

is not deterministic, we run CPLEX also in its nondeterministic ("opportunistic") mode.

DDX10 is implemented using X10 2.3.1 [158] and compiled using the C++ back-end with g++ 4.4.5. [2] For all experiments with DDX10, we used the following values of the parameters of the parallelization scheme: $maxPQueueLength = 5.5 \times 10^9$ (determined based on the available memory on the machine storing the global queue), $maxLocalPoolSize = 1000, c = 10, \bar{c} = 1.0, c^* = 2.0, d = 0.1$, and $d^* = 100$. The maximum width W for the BDD generated at each subproblem was set to be the number of free variables (i.e., the number of active vertices) in the state of the BDD node that generated the subproblem. The type of exact cut used in the branch-and-bound algorithm for the experiments was the *frontier cut* [22]. These values and parameters were chosen based on experimentation on our cluster with a few representative instances, keeping in mind their overall impact on load balancing and pruning as discussed earlier.

DDX10 Versus Parallel MIP

The comparison between DDX10 and IBM ILOG CPLEX 12.5.1 was conducted on 2.3 GHz AMD Opteron 6134 machines with 32 cores, 64 GB RAM, 512 KB L2 cache, and 12 MB L3 cache.

To draw meaningful conclusions about the scaling behavior of CPLEX vs. DDX10 as the number w of workers is increased, we start by selecting problem instances where both approaches exhibit comparable performance in the sequential setting. To this end, we report an empirical evaluation on random instances with 170 vertices and six graph densities $\rho = 0.19, 0.21, 0.23, 0.25, 0.27$, and 0.29. For each ρ, we generated five random graphs, obtaining a total of 30 problem instances. For each pair (ρ, w) with w being the number of workers, we aggregate the runtime over the five random graphs using the geometric mean.

Figure 6.9 summarizes the result of this comparison for $w = 1, 2, 4, 16$, and 32. As we see, CPLEX and DDX10 display comparable performance for $w = 1$ (the leftmost data points). While the performance of CPLEX varies relatively little as a function of the graph density ρ, that of DDX10 varies more widely. As observed earlier in this section for the sequential case, BDD-based branch-and-bound performs better on higher-density graphs than sparse graphs. Nevertheless,

[2] The current version of DDX10 may be downloaded from `http://www.andrew.cmu.edu/user/vanhoeve/mdd`.

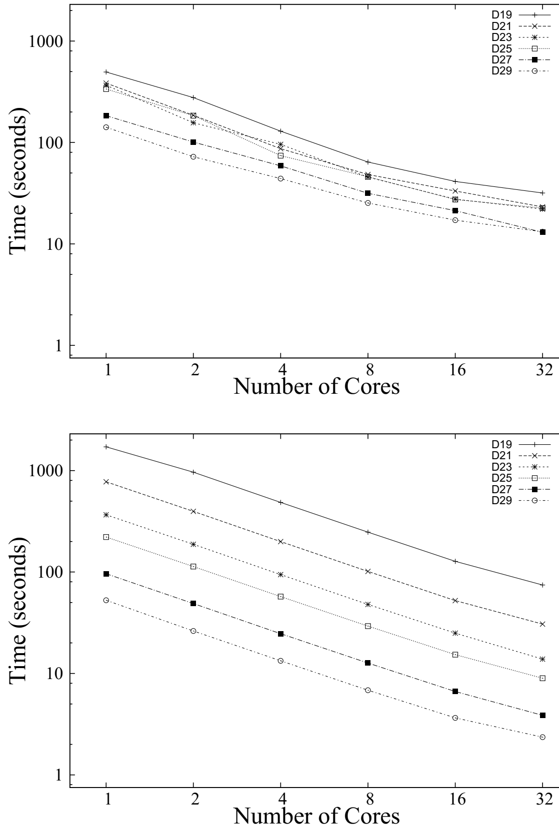


Fig. 6.9 Performance of CPLEX (above) and DDX10 (below), with one curve for each graph density ρ shown in the legend as a percentage. Both runtime (y-axis) and number of cores (x-axis) are on logarithmic scale.

the performance of the two approaches when $w = 1$ is in a comparable range for the observation we want to make, which is the following: *DDX10 scales more consistently than CPLEX when invoked in parallel* and also retains its advantage on higher-density graphs. For $\rho > 0.23$, DDX10 is clearly exploiting parallelism better than CPLEX. For example, for $\rho = 0.29$ and $w = 1$, DDX10 takes about 80 seconds to solve the instances while CPLEX needs about 100 seconds—a modest performance ratio of 1.25. This same performance ratio increases to 5.5 when both methods use $w = 32$ workers.

Table 6.2 Runtime (seconds) of DDX10 on DIMACS instances. Timeout = 1,800.

Instance	n	Density	1 core	4 cores	16 cores	64 cores	256 cores
hamming8-4.clq	256	0.36	25.24	7.08	2.33	1.32	0.68
brock200_4.clq	200	0.34	33.43	9.04	2.84	1.45	1.03
san400_0.7_1.clq	400	0.30	33.96	9.43	4.63	1.77	0.80
p_hat300-2.clq	300	0.51	34.36	9.17	2.74	1.69	0.79
san1000.clq	1000	0.50	40.02	12.06	7.15	2.15	9.09
p_hat1000-1.clq	1000	0.76	43.35	12.10	4.47	2.84	1.66
sanr400_0.5.clq	400	0.50	77.30	18.10	5.61	2.18	2.16
san200_0.9_2.clq	200	0.10	93.40	23.72	7.68	3.64	1.65
sanr200_0.7.clq	200	0.30	117.66	30.21	8.26	2.52	2.08
san400_0.7_2.clq	400	0.30	234.54	59.34	16.03	6.05	4.28
p_hat1500-1.clq	1500	0.75	379.63	100.3	29.09	10.62	25.18
brock200_1.clq	200	0.25	586.26	150.3	39.95	12.74	6.55
hamming8-2.clq	256	0.03	663.88	166.49	41.80	23.18	14.38
gen200_p0.9_55.clq	200	0.10	717.64	143.90	43.83	12.30	6.13
C125.9.clq	125	0.10	1,100.91	277.07	70.74	19.53	8.07
san400_0.7_3.clq	400	0.30	–	709.03	184.84	54.62	136.47
p_hat500-2.clq	500	0.50	–	736.39	193.55	62.06	23.81
p_hat300-3.clq	300	0.26	–	–	1,158.18	349.75	172.34
san400_0.9_1.clq	400	0.10	–	–	1,386.42	345.66	125.27
san200_0.9_3.clq	200	0.10	–	–	–	487.11	170.08
gen200_p0.9_44.clq	200	0.10	–	–	–	1,713.76	682.28
sanr400_0.7.clq	400	0.30	–	–	–	–	1,366.98
p_hat700-2.clq	700	0.50	–	–	–	–	1,405.46

Parallel Versus Sequential Decision Diagrams

The two experiments reported in this section were conducted on a larger cluster, with 13 of 3.8 GHz Power7 machines (CHRP IBM 9125-F2C) with 32 cores (4-way SMT for 128 hardware threads) and 128 GB of RAM. The machines are connected via a network that supports the PAMI message passing interface [106], although DDX10 can also be easily compiled to run using the usual network communication with TCP sockets. We used 24 workers on each machine, using as many machines as necessary to operate w workers in parallel.

Random Instances

The first experiment reuses the random MISP instances introduced in the previous section, with the addition of similar but harder instances on graphs with 190 vertices, resulting in 60 instances in total.

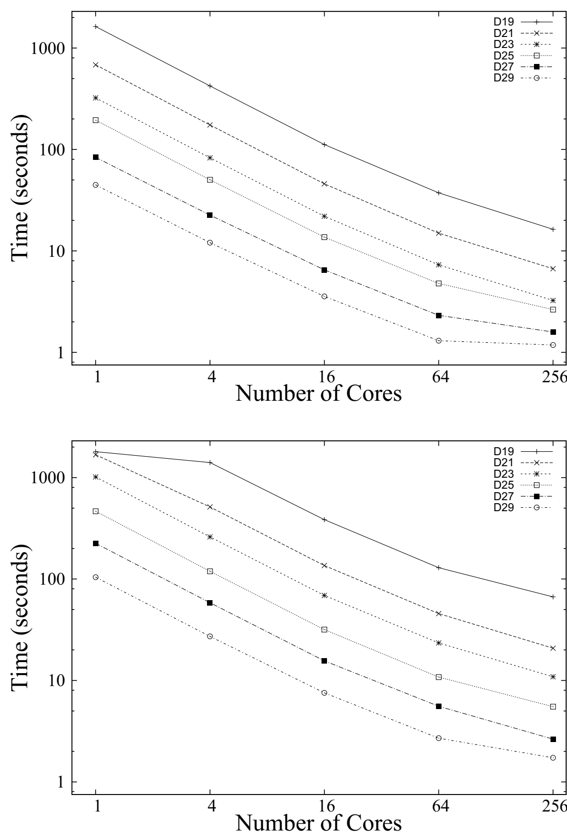


Fig. 6.10 Scaling behavior of DDX10 on MISP instances with 170 (above) and 190 (below) vertices, with one curve for each graph density ρ shown in the legend as a percentage. Both runtime (y-axis) and number of cores (x-axis) are on logarithmic scale.

As Fig. 6.10 shows, DDX10 scales near-linearly up to 64 cores and still very well up to 256 cores. The slight degradation in performance when going to 256 cores is more apparent for the higher-density instances (lower curves in the plots), which do not have much room left for linear speedups as they need only a couple of seconds to be solved with 64 cores. For the harder instances (upper curves), the scaling is still satisfactory even if not linear. As noted earlier, coming anywhere close to near-linear speedups for complex combinatorial search and optimization methods has been remarkably hard for SAT and MIP. These results show that parallelization of BDD based branch-and-bound can be much more effective.

Table 6.3 Number of nodes in multiples of 1,000 processed (#No) and pruned (#Pr) by DDX10 as a function of the number of cores. Same setup as in Table 6.2.

Instance	1 core		4 cores		16 cores		64 cores		256 cores	
	#No	#Pr	#No	#Pr	#No	#Pr	#No	#Pr	#No	#Pr
hamming8-4.clq	43	0	42	0	40	0	32	0	41	0
brock200_4.clq	110	42	112	45	100	37	83	30	71	25
san400_0.7_1.clq	7	1	8	1	6	0	10	1	14	1
p_hat300-2.clq	80	31	74	27	45	11	46	7	65	12
san1000.clq	29	16	50	37	18	4	13	6	28	6
p_hat1000-1.clq	225	8	209	0	154	1	163	1	206	1
sanr400_0.5.clq	451	153	252	5	354	83	187	7	206	5
san200_0.9_2.clq	22	0	20	0	19	0	18	1	25	0
sanr200_0.7.clq	260	3	259	5	271	17	218	4	193	6
san400_0.7_2.clq	98	2	99	5	112	21	147	67	101	35
p_hat1500-1.clq	1,586	380	1,587	392	1,511	402	962	224	1,028	13
brock200_1.clq	1,378	384	1,389	393	1,396	403	1,321	393	998	249
hamming8-2.clq	45	0	49	0	49	0	47	0	80	0
gen200_p0.9_55.clq	287	88	180	6	286	90	213	58	217	71
C125.9.clq	1,066	2	1,068	0	1,104	38	1,052	13	959	19
san400_0.7_3.clq	–	–	2,975	913	2,969	916	2,789	779	1,761	42
p_hat500-2.clq	–	–	2,896	710	3,011	861	3,635	1,442	2,243	342
p_hat300-3.clq	–	–	–	–	18,032	4,190	17,638	3,867	15,852	2,881
san400_0.9_1.clq	–	–	–	–	2,288	238	2,218	207	2,338	422
san200_0.9_3.clq	–	–	–	–	–	–	9,796	390	10,302	872
gen200_p0.9_44.clq	–	–	–	–	–	–	43,898	5,148	45,761	7,446
sanr400_0.7.clq	–	–	–	–	–	–	–	–	135,029	247
p_hat700-2.clq	–	–	–	–	–	–	–	–	89,845	8,054

DIMACS Instances

The second experiment is on the DIMACS instances used by [22], where it was demonstrated that sequential BDD-based branch-and-bound has complementary strengths compared with sequential CPLEX and outperforms the latter on several instances, often the ones with higher graph density ρ. We consider here the subset of instances that take at least 10 seconds (on our machines) to solve using sequential BDDs and omit any that cannot be solved within the time limit of 1800 seconds (even with 256 cores). The performance of DDX10 with $w = 1, 4, 16, 64$, and 256 is reported in Table 6.2, with rows sorted by hardness of instances.

These instances represent a wide range of graph size, density, and structure. As we see from the table, DDX10 is able to scale very well to 256 cores. Except for three instances, it is significantly faster on 256 cores than on 64 cores, despite the substantially larger communication overhead for workload distribution and bound sharing.

Table 6.3 reports the total number of nodes processed through the global queue, as well as the number of nodes pruned due to bounds communicated by the workers.[3] Somewhat surprisingly, the number of nodes processed does not increase by much compared with the sequential case, despite the fact that hundreds of workers start processing nodes in parallel without waiting for potentially improved bounds which might have been obtained by processing nodes sequentially. Furthermore, the number of pruned nodes also stays steady as w grows, indicating that bound communication is working effectively. This provides insight into the amiable scaling behavior of DDX10 and shows that it is able to retain sufficient global knowledge even when executed in a distributed fashion.

[3] Here we do not take into account the number of nodes added to local pools, which is usually a small fraction of the number of nodes processed by the global pool.

Chapter 7
Variable Ordering

Abstract One of the most important parameters that determines the size of a decision diagram is the variable ordering. In this chapter we formally study the impact of variable ordering on the size of exact decision diagrams for the maximum independent set problem. We provide worst-case bounds on the size of the exact decision diagram for particular classes of graphs. For general graphs, we show that the size is bounded by the Fibonacci numbers. Lastly, we demonstrate experimentally that variable orderings that produce small exact decision diagrams also produce better bounds from relaxed decision diagrams.

7.1 Introduction

The ordering of the vertices plays an important role in not only the size of exact decision diagrams, but also in the bound obtained by DD-based relaxations and restrictions. It is well known that finding orderings that minimize the size of DDs (or even improving on a given ordering) is NP-hard [58, 35]. We found that the ordering of the vertices is the single most important parameter in creating small-width exact DDs and in proving tight bounds via relaxed DDs.

In this chapter we analyze how the combinatorial structure of a problem can be exploited to develop variable orderings that bound the size of the DD representing its solution space. We will particularly focus on the maximum independent set problem (MISP) for our analysis, first described in Section 3.5. Given a graph $G = (V,E)$ with a vertex set V, an independent set I is a subset $I \subseteq V$ such that no two vertices in I are connected by an edge in E, i.e., $(u,v) \notin E$ for any distinct $u,v \in I$. If we

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_7

associate weights with each vertex $j \in V$, the MISP asks for a maximum-weight independent set of G. Since variables are binaries, the resulting diagram is a binary decision diagram (BDD).

Different orderings can yield exact BDDs with dramatically different widths. For example, Fig. 7.1 shows a path on six vertices with two different orderings given by $x_1,\ldots,x_6$ and $y_1,\ldots,y_6$. In Fig. 7.2(a) we see that the vertex ordering $x_1,\ldots,x_6$ yields an exact BDD with width 1, while in Fig. 7.2(b) the vertex ordering $y_1,\ldots,y_6$ yields an exact BDD with width 4. This last example can be extended to a path with $2n$ vertices, yielding a BDD with a width of 2^{n-1}, while ordering the vertices according to the order in which they lie on the paths yields a BDD of width 1.

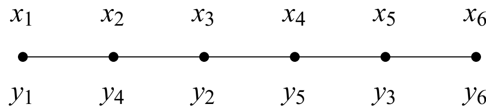


Fig. 7.1 Path graph.

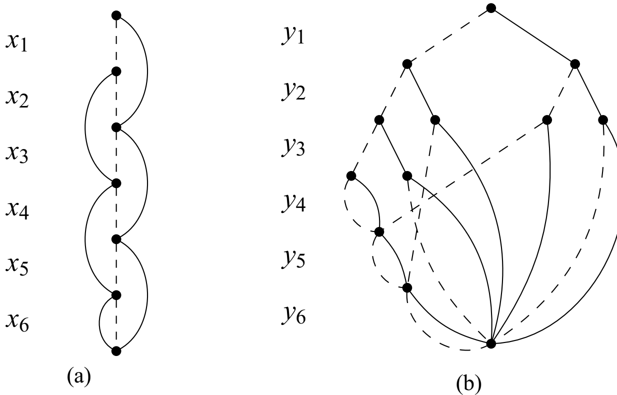


Fig. 7.2 (a) Variable order that results in a small reduced BDD. (b) Variable order that results in a larger reduced BDD.

Our study focuses on studying variable orderings for the layers of a BDD representing the set of feasible solutions to a MISP instance. For particular classes of graphs, variable orderings are given that can be used to provide worst-case bounds on the width of exact BDDs [24, 89]. This is followed by the description of a family of variable orderings for which the exact BDD is bounded by the Fibonacci numbers. Based on this analysis, various heuristic orderings for relaxed BDDs are suggested,

which operate on the assumption that an ordering that results in a small-width exact reduced BDD also results in a relaxed BDD with a strong optimization bound.

Even though these orderings are specific to the maximum independent set problem, they indicate novel *look-ahead* ordering heuristics that are applicable to any combinatorial optimization problem. Recent work has also extended the results to more general independent systems [89], relating the size of a BDD with the bandwidth of the constraint matrix.

7.2 Exact BDD Orderings

Let $E(u)$ be the state associated with a node u, and let $S(L_j)$ be the set of states on nodes in L_j, $S(L_j) = \cup_{u \in L_j} E(u)$. To bound the width of a given layer j, we need only count the number of states that may arise from independent sets on $\{v_1, \ldots, v_{j-1}\}$. This is because each layer will have one and only one node for each possible state, and so there is a one-to-one correspondence between the number of states and the size of a layer.

It is assumed for the remainder of this chapter that the form of the decision diagram used is not a *zero-compressed* decision diagram, as is shown in the figures above. The bounds can be slightly improved should zero-compressed decision diagrams be employed, but for ease of exposition and clarity the use of BDDs is assumed.

Theorem 7.1. *Let $G = (V, E)$ be a clique. Then, for any ordering of the vertices, the width of the exact reduced BDD will be 2.*

Proof. Consider any layer j. The only possible independent sets on $\{v_1, \ldots, v_{j+1}\}$ are $\emptyset$ or $\{v_i\}, i = 1, \ldots, j-1$. For the former, $E(\emptyset \mid \{v_j, \ldots, v_n\}) = \{v_j, \ldots, v_n\}$ and for the latter, $E(\{v_i\} \mid \{v_j, \ldots, v_n\}) = \emptyset$, establishing the bound. □

Theorem 7.2. *Let $G = (V, E)$ be a path. Then, there exists an ordering of the vertices for which the width of the exact reduced BDD will be 2.*

Proof. Let the ordering of the vertices be given by the positions in which they appear in the path. Consider any layer j. Of the remaining vertices in G, namely $\{v_j, \ldots, v_n\}$, the only vertex with any adjacencies to $\{v_1, \ldots, v_{j-1}\}$ is v_j. Therefore, for any independent set $I \subseteq \{v_1, \ldots, v_{j-1}\}$, $E(I \mid \overline{V}_{j-1})$ will be either $\{v_j, \ldots, v_n\}$

(when $v_{j-1} \notin I$) or $\{v_{j+1},\ldots,v_n\}$ (when $v_{j-1} \in I$). Therefore there can be at most two states in any given layer. □

Theorem 7.3. *Let $G = (V,E)$ be a cycle — a graph isomorphic, for some n, to the graph with $V = \{v_1,\ldots,v_n\}$ and $E = \big\{\{v_j,v_{j+1}\} : j = 1,\ldots,n-1\big\} \cup \{v_1,v_n\}$. There exist orderings of the vertices for which the width of the exact reduced BDD will be 4.*

Proof. Using the ordering defined by V above, let u be any node in B in layer j and I the independent set it corresponds to. Four cases are possible and will define $E(u)$, implying the width of 4:

$$E(u) = \begin{cases} \{v_j,v_{j+1},\ldots,v_n\} & v_1,v_{j-1} \notin I \\ \{v_j,v_{j+1},\ldots,v_{n-1}\} & v_1 \in I, v_{j-1} \notin I \\ \{v_{j+1},\ldots,v_n\} & v_1 \notin I, v_{j-1} \in I \\ \{v_{j+1},v_{j+1},\ldots,v_{n-1}\} & v_1,v_{j-1} \in I. \ \square \end{cases}$$

Theorem 7.4. *Let $G = (V,E)$ be a complete bipartite graph — a graph for which the vertex set V can be partitioned into two sets V_1,V_2 so that*

$$E \subseteq \{\{v_1,v_2\} : v_1 \in V_1, v_2 \in V_2\}.$$

There exist orderings of the vertices for which the width of the exact reduced BDD will be 2.

Proof. Let V_1,V_2 be the two partitions of V providing the necessary conditions for the graph being bipartite, and let the variables be ordered so that $V_1 = \{v_1,\ldots,v_{|V_1|}\}$ and $V_2 = \{v_{|V_1|+1},\ldots,v_n\}$. Let B be the exact reduced BDD in this ordering, u any node in B, and I the independent set induced by u.

Let j be the layer of u. If I does not contain any vertex in V_1 then $E(u) = \{v_j,\ldots,v_n\}$. Otherwise, I contains only vertices in the first shore. Therefore, if $j \le |V_1|, E(u) = \{v_j,\ldots,v_{|V_1|}\}$ and if $j > |V_1|, E(u) = \emptyset$. As these are the only possibilities, the width of any of these layers is at most 2. □

The above also implies, for example, that star graphs have the same width of 2.

We now consider interval graphs, that is, graphs that are isomorphic to the intersection graph of a multiset of intervals on the real line. Such graphs have vertex orderings $v_1,\ldots v_n$ for which each vertex v_i is adjacent to the set of vertices

$v_{a_i}, v_{a_i+1}, \dots v_{i-1}, v_{i+1}, \dots, v_{b_i}$ for some a_i, b_i. We call such an ordering an *interval ordering* for G. Note that paths and cliques, for example, are contained in this class of graphs. In addition, note that determining whether or not an interval ordering exists (and finding such an ordering) can be done in linear time in n.

Theorem 7.5. *For any interval graph, any interval ordering $v_1, \dots, v_n$ yields an exact reduced BDD for which the width will be no larger than n, the number of vertices in G.*

Proof. Let $T_k = \{v_k, \dots, v_n\}$. It is shown here that, for any u in the exact BDD created using any interval ordering, $E(u) = T_k$ for some k.

Each node u corresponds to some independent set in G. Fix u and let V' contained in $\{v_1, \dots, v_{k-1}\}$ be the independent set in G that it corresponds to. Let $\tilde{b}$ be the maximum right limit of the intervals corresponding to the vertices in V'. Let $\tilde{a}$ be the minimum left limit of the intervals corresponding to the variables in $\{v_1, \dots, v_{k-1}\}$, which is larger than $\tilde{b}$, and k' the index of the graph vertex with this limit (i.e., for which $a_{k'} = \tilde{a}$). $E(u) = V_{k'}$, and since u was arbitrary, the theorem follows, and $\omega(B) \leq n$. □

Theorem 7.6. *Let $G = (V, E)$ be a tree. Then, there exists an ordering of the vertices for which the width of the exact reduced BDD will be no larger than n, the number of vertices in G.*

Proof. We proceed by induction on n. For the base case, a tree with 2 vertices is a path, which we already know has width 2. Now let T be a tree on n vertices. Any tree on n vertices contains a vertex v for which the connected components $C_1, \dots, C_k$ created upon deleting v from T have sizes $|C_i| \leq \frac{n}{2}$ [103]. Each of these connected components are trees with fewer than $\frac{n}{2}$ vertices, so by induction, there exists an ordering of the vertices on each component C_i for which the resulting BDD B_i will have width $\omega(B_i) \leq \frac{n}{2}$. For component C_i, let $v_1^i, \dots, v_{|C_i|}^i$ be an ordering achieving this width.

Let the final ordering of the vertices in T be $v_1^1, \dots, v_{|C_1|}^1, v_1^2, \dots, v_{|C_k|}^k, v$, which we use to create BDD B for the set of independent sets in T. Consider layer $\ell \leq n-1$ of B corresponding to vertex v_j^i. We claim that the only possible states in $S(\ell)$ are $s \cup C_{i+1} \cup \dots \cup C_k$ and $s \cup C_{i+1} \cup \dots \cup C_k \cup \{v\}$, for $s \in S^i(j)$, where $S^i(j)$ is the set of states in BDD B_i in layer j. Take any independent set on the vertices $I \subseteq \{v_1^1, \dots, v_{|C_1|}^1, v_1^2, \dots, v_{j-1}^i\}$. All vertices in I are independent of the vertices in $C_{i+1}, \dots, C_k$, and so $E(I \mid \{v_j^i, \dots, v_{|C_i|}^i\} \cup C_{i+1} \cup \dots \cup C_k) \supseteq C_{i+1} \cup \dots \cup C_k$. Now,

consider $I_i = I \cap C_i$. I_i is an independent set in the tree induced on the variables in C_i and so it will correspond to some path in B_i from the root of that BDD to layer j, ending at some node u. The state s of node u contains all of the vertices $\{v^i_j, \ldots, v^i_{|C_i|}\}$ that are independent of all vertices in I_i. As $v^i_1, \ldots, v^i_{j-1}$ are the only vertices in the ordering up to layer ℓ in B that have adjacencies to any vertices in C_i, we see that the set of vertices in the state of I from component C_i are exactly s. Therefore, $E(I \mid \{v^i_j, \ldots, v^i_{|C_i|}\} \cup C_{i+1} \cup \cdots \cup C_k) \supseteq s \cup C_{i+1} \cup \cdots \cup C_k$. The only remaining vertex that may be in the state is v, finishing the claim. Therefore, as the only possible states on layer ℓ are $s \cup C_{i+1} \cup \cdots \cup C_k$ and $s \cup C_{i+1} \cup \cdots \cup C_k \cup \{v\}$, for $s \in S^i(j)$, we see that $\omega_\ell \le \frac{n}{2} \cdot 2 = n$, as desired. The layer remaining to bound is L_n, which contains $\{v\}$ and $\emptyset$. $\square$

Theorem 7.7. *Let $G = (V, E)$ be any graph. There exists an ordering of the vertices for which $\omega_j \le F_{j+1}$, where F_k is the kth Fibonacci number.*

Theorem 7.7 provides a bound on the width of the exact BDD for any graph. The importance of this theorem goes further than the actual bound provided on the width of the exact BDD for any graph. First, it illuminates another connection between the Fibonacci numbers and the family of independent sets of a graph, as investigated throughout the literature graph theory (see, for example, [38, 67, 57, 159]). In addition to this theoretical consideration, the underlying principles in the proof provide insight into what heuristic ordering for the vertices in a graph could lead to BDDs with small width. The ordering inspired by the underlying principle in the proof yields strong relaxation BDDs.

Proof (Proof of Theorem 7.7). Let $P = P^1, \ldots, P^k$, $P^i = \{v^1_1, \ldots, v^1_{i_k}\}$ be a *maximal path decomposition* of the vertices of G, where by a maximal path decomposition we mean a set of paths that partition V satisfying that v^i_1 and $v^i_{i_k}$ are not adjacent to any vertices in $\cup_{j=i+1}^k P^j$. Hence, P^i is a maximal path (in that no vertices can be appended to the path) in the graph induced by the vertices not in the paths, $P^1, \ldots, P^{i-1}$.

Let the ordering of the vertices be given by $v^1_1, \ldots, v^1_{i_1}, v^2_1, \ldots, v^k_{i_k}$, in which, ordered by the paths and by the order they appear on the paths. Let the vertices also be labeled, in this order, by $y_1, \ldots, y_n$.

We proceed by induction, showing that, if layers L_j and L_{j+1} have widths ω_j and ω_{j+1}, respectively, then the width of layer L_{j+3} is bounded by $\omega_j + 2 \cdot \omega_{j+1}$, thereby proving that each layer L_j is bounded by F_{j+1} for every layer $j = 1, \ldots, n+1$, since $F_{j+3} = F_j + 2 \cdot F_{j+1}$.

First we show that L_4 has width bounded by $F_5 = 5$. We can assume that G is connected and has at least 4 vertices, so that P_1 has at least 3 vertices. $\omega_1 = 1$. Also, $\omega_2 = 2$, with layer L_2 having nodes u_1^2, u_2^2 arising from the partial solutions $I = \emptyset$ and $I = \{w_1\}$, respectively. The corresponding states will be $E(u_1^2) = V \backslash \{y_1\}$ and $E(u_2^2) = V \backslash (\{y_1\} \cup N(y_1))$. Now, consider layer L_3. The partial solution ending at node $E(u_2^2)$ cannot have y_2 added to the independent set because y_2 does not appear in $E(u_2^2)$ since $y_2 \in N(w_1)$. Therefore, there will be exactly three outgoing arcs from the nodes in L_2. If no nodes are combined on the third layer, there will be 3 nodes $u_i^3, i = 1,2,3$ with states $E(u_1^3) = V \backslash \{y_1, y_2\}$, $E(u_2^3) = V \backslash (\{y_1, y_2\} \cup N(y_2))$, and $E(u_3^3) = V \backslash (\{y_1, y_2\} \cup N(y_1))$. Finally, as P^1 has length at least 3, vertex y_3 is adjacent to y_2. Therefore, we cannot add y_3 under node u_2^3, so layer 4 will have width at most 5, finishing the base case.

Now let the layers of the partially constructed BDD be given by $L_1, \ldots, L_j, L_{j+1}$ with corresponding widths $\omega_i, i = 1, \ldots, j+1$. We break down into cases based on where y_{j+1} appears in the path that it belongs to in P, as follows:

Case 1: y_{j+1} is the last vertex in the path that it belongs to. Take any node $u \in L_{j+1}$ and its associated state $E(u)$. Including or not including y_{j+1} results in state $E(u) \backslash \{y_{j+1}\}$ since y_{j+1} is independent of all vertices $y_i, i \geq j+2$. Therefore, $\omega_{j+2} \leq \omega_{j+1}$ since each arc directed out of u will be directed at the same node, even if the zero-arc and the one-arc are present. And, since in any BDD $\omega_k \leq 2 \cdot \omega_{k-1}$, we have $\omega_{j+3} \leq 2 \cdot \omega_{j+2} \leq 2 \cdot \omega_{j+1} < \omega_j + 2 \cdot \omega_{j+1}$.

Case 2: y_{j+1} is the first vertex in the path that it belongs to. In this case, y_j must be the last vertex in the path that it belongs to. By the reasoning in case 1, it follows that $\omega_{j+1} \leq \omega_j$. In addition, we can assume that y_{j+1} is not the last vertex in the path that it belongs to because then we are in case 1. Therefore, y_{j+2} is in the same path as y_{j+1} in P. Consider L_{j+2}. In the worst case, each node in L_{j+1} has y_{j+1} in its state so that $\omega_{j+2} = 2 \cdot \omega_{j+1}$. But, any node arising from a one-arc will not have y_{j+2} in its state. Therefore, there are at most ω_{j+1} nodes in L_{j+2} with y_{j+2} in their states and at most ω_{j+1} nodes in L_{j+2} without y_{j+2} in their states. For the set of nodes without y_{j+2} in their states, we cannot make a one-arc, showing that $\omega_{j+3} \leq \omega_{j+2} + \omega_{j+1}$. Therefore, we have $\omega_{j+3} \leq \omega_{j+1} + \omega_{j+2} \leq 3 \cdot \omega_{j+1} \leq \omega_j + 2 \cdot \omega_{j+1}$.

Case 3: y_{j+1} is not first or last in the path that it belongs to. As in case 2, $\omega_{j+1} \leq 2 \cdot \omega_j$, with at most ω_j nodes on layer L_{j+1} with w_{j+2} in its corresponding state label. Therefore, L_{j+2} will have at most ω_j more nodes in it than layer L_{j+1}.

As the same holds for layer L_{j+3}, in that it will have at most ω_{j+1} more nodes in it than layer L_{j+2}, we have $\omega_{j+3} \leq \omega_{j+2} + \omega_{j+1} \leq \omega_{j+1} + \omega_j + \omega_{j+1} = \omega_j + 2 \cdot \omega_{j+1}$, as desired, finishing the proof. □

We note here that, using instance `C2000.9` from the DIMACS benchmark set,[1] a maximal path decomposition ordering of the vertices yields widths approximately equal to the Fibonacci numbers, as seen in Table 7.1.

Table 7.1 Widths of exact BDD for `C.2000.9`

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	⋯
w_j	1	2	3	5	8	13	21	31	52	65	117	182	299	481	624	⋯
$Fib(j+1)$	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	⋯

7.3 Relaxed BDD Orderings

In this section we provide heuristic orderings for the vertices to be used during the top-down compilation of relaxation BDDs. The orderings are suggested based on the theorems proved in the previous sections, with the idea that, by examining simple structured problems, we can gain intuition as to what is controlling the width of the exact BDD for general graphs, hopefully yielding tighter upper bounds.

Maximal Path Decomposition (MPD). As show in Theorem 7.7, such an ordering yields an exact BDD with width bounded by the Fibonacci numbers, yielding a theoretical worst-case bound on the width for any instance. This ordering can be precomputed in worst-case time complexity $O(|V| + |E|)$. We note that different maximal path decompositions may yield different sized BDDs.

Minimum Number of States (MIN). In this ordering, we select the next vertex in the BDD as the vertex which appears in the fewest states of the layer we are currently building. The driving force behind the proof of Theorem 7.7 is that when constructing a layer, if a vertex does not belong to the state of a node on a previous layer, we cannot include this vertex, i.e., we cannot add a one-arc, only the zero-arc. This suggests that selecting a variable appearing the fewest number of times in the

[1] `http://dimacs.rutgers.edu/Challenges/`

states on a layer will yield a small-width BDD. The worst-case time complexity to perform this selection is $O(W|V|)$ per layer.

k-Look Ahead Ordering (kLA). This ordering can be employed for any binary optimization. In *1LA*, after selecting the first j vertices and constructing the top $j+1$ layers, the next chosen vertex is the one that yields the smallest width for layer $j+2$ if it were selected next. This procedure can be generalized for arbitrary $k < n$ by considering subsets of yet to be selected vertices. The worst-case running time for selecting a vertex can be shown to be $O(\binom{n}{k} \cdot W|V|^2 \log|W|)$ per layer.

For general k we can proceed as follows. Let $X = \{x_1, \ldots, x_n\}$. We begin by selecting every possible set of k variables for X. For each set S, we build the exact BDD using any ordering of the variables in S, yielding an exact BDD up to layer $k+1$. We note that it suffices to just consider sets of variables as opposed to permutations of variables because constructing a partial exact BDD using any ordering in S would yield the same number of states (and hence the same width) for layer $k+1$. We then select the set which yields the fewest number of nodes in the resulting partially constructed BDD, using the variable in S that yields the smallest width of layer 2 as the first variable in the final ordering.

Continuing in this fashion, for $j \geq 2$, we select every set of the unselected variables of size $k = \min k, n-j$ and construct the exact BDD if these were the next selected vertices. For the set S that achieves the minimum width of layer L_{j+k+1}, we choose the variable in S that minimizes the width if it were to be selected as the next vertex, and continue until all layers are built. The worst case running time for selecting a vertex can be shown to be $O\left(\binom{n}{k} \cdot W|V|^2 \log|W|\right)$ per layer.

7.4 Experimental Results

Our experiments focus on the complement graphs of the well-known DIMACS problem set for the maximum clique problem, which can be obtained by accessing `http://dimacs.rutgers.edu/Challenges/`. The experiments ran on an Intel Xeon E5345 with 8 GB RAM. The BDD was implemented in C++.

7.4.1 Exact BDDs for Trees

The purpose of the first set of experiments is to demonstrate empirically that variable orderings potentially play a key role in the width of exact BDDs representing combinatorial optimization problems. To this end, we have selected a particular graph structure, namely trees, for which we can define an ordering yielding a polynomial bound on its width (Theorem 7.6). We then compare the ordering that provides this bound with a set of randomly generated orderings. We also compare with the MPD heuristic, which has a known bound for general graphs according to Theorem 7.7. The trees were generated from the benchmark problems `C125.9`, `keller4`, `c-fat100-1`, `p_hat300-1`, `brock200_1`, and `san200_0.7_1` by selecting 5 random trees each on 50 vertices from these graphs. The tree-specific ordering discussed in Theorem 7.6 is referred to as the *CV* (due to the computation of cut-vertices in the corresponding proof). We generated exact BDDs using 100 uniform-random orderings for each instance, and report the minimum, average, and maximum obtained widths.

The results are shown in Table 7.2. In all cases, none of the 100 random orderings yielded exact BDDs with width smaller than the ones generated from the CV or MPD orderings. Moreover, the average was consistently more than an order of magnitude worse than either of the structured orderings. This confirms that investigating variable orderings can have a substantial effect on the width of the exact BDDs produced for independent set problems. In addition, we see that also across all instances, the CV ordering, which is specific to trees, outperforms the MPD ordering that can be applied to general graphs, suggesting that investigating orderings specific to particular classes of instances can also have a positive impact on the width of exact BDDs.

7.4.2 Exact BDD Width Versus Relaxation BDD Bound

The second set of experiments aims at providing empirical evidence to the hypothesis that a problem instance with a smaller exact BDD results in a relaxation BDD that yields a tighter bound. The instances in this test were generated as follows: We first selected five instances from the DIMACS benchmark: `brock200_1`, `gen200_p.0.9_55`, `keller4`, `p_hat300-2`, and `san200_0.7_1`. Then, we uniformly at random extracted 5 connected induced subgraphs with 50 vertices for

Table 7.2 Random trees

Instance	*Min*	*Avg*	*Max*	*CV*	*MPD*	*Instance*	*Min*	*Avg*	*Max*	*CV*	*MPD*
brock200_1.t-1	2336	22105.1	116736	16	160	C125.9.t-1	768	7530.72	24576	12	228
brock200_1.t-2	672	8532.92	86016	16	312	C125.9.t-2	1600	19070	131072	12	528
brock200_1.t-3	672	7977.92	28608	8	120	C125.9.t-3	1024	8348.04	30720	12	288
brock200_1.t-4	2880	17292.9	67200	16	132	C125.9.t-4	736	4279.62	16704	16	312
brock200_1.t-5	1200	12795.2	55680	8	54	C125.9.t-5	480	18449.3	221184	16	120
c-fat200-1.t-1	896	17764.3	221184	8	112	keller4.t-1	952	9558.76	115200	8	248
c-fat200-1.t-2	1152	10950.9	55040	16	144	keller4.t-2	768	8774.12	71680	12	444
c-fat200-1.t-3	2048	23722.6	150528	10	72	keller4.t-3	2688	16942.1	74240	10	40
c-fat200-1.t-4	624	5883.96	46656	12	180	keller4.t-4	2048	14297.8	77440	16	368
c-fat200-1.t-5	864	7509.66	27648	10	480	keller4.t-5	720	11401.8	73728	8	288
p_hat300-1.t-1	792	15149.3	54720	10	200	san200_0.7_1.t-1	1920	22771.2	139776	10	28
p_hat300-1.t-2	1280	14618.5	86016	16	192	san200_0.7_1.t-2	1024	7841.42	44160	12	92
p_hat300-1.t-3	624	11126.6	69120	12	138	san200_0.7_1.t-3	768	8767.76	36864	8	88
p_hat300-1.t-4	1152	13822.9	73984	16	74	san200_0.7_1.t-4	960	9981.28	43008	16	84
p_hat300-1.t-5	1536	16152	82944	14	160	san200_0.7_1.t-5	1536	9301.92	43008	12	288

each instance, which is approximately the largest graph size for which the exact BDD can be built within our memory limits.

The tests are described next. For each instance and all orderings MPD, MIN, random, and 1LA, we collected the width of the exact BDD and the bound obtained by a relaxation BDD with a maximum width of 10 (the average over 100 orderings for the random procedure). This corresponds to sampling different exact BDD widths and analyzing their respective bounds, since distinct variable orderings may yield BDDs with very different exact widths.

Figure 7.3 presents a scatter plot of the derived upper bound as a function of the exact widths in log-scale, also separated by the problem class from which the instance was generated. Analyzing each class separately, we observe that the bounds and width increase proportionally, reinforcing our hypothesis. In particular, this proportion tends to be somewhat constant, that is, the points tend to a linear curve for each class. We notice that this shape has different slopes according to the problem class, hence indicating that the effect of the width might be more significant for certain instances.

In Fig. 7.4 we plot the bound as a function of the exact width for a single random instance extracted from `san200_0.7_1`. In this particular case, we applied a procedure that generated 1000 exact BDDs with a large range of widths: the minimum observed BDD width was 151 and the maximum was 27,684, and the widths were approximately uniformly distributed in this interval. We then computed the corresponding upper bounds for a relaxed BDD, constructed using the orderings described above, with width 10. The width is given in a log-scale. The figure also shows a strong correlation between the width and the obtained bound, analogous to

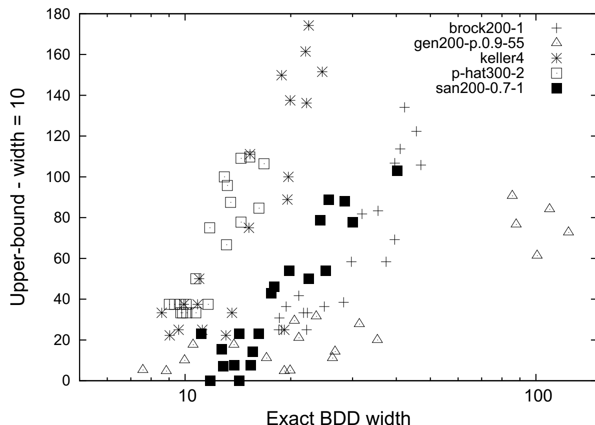


Fig. 7.3 Bound of relaxation BDD vs. exact BDD width.

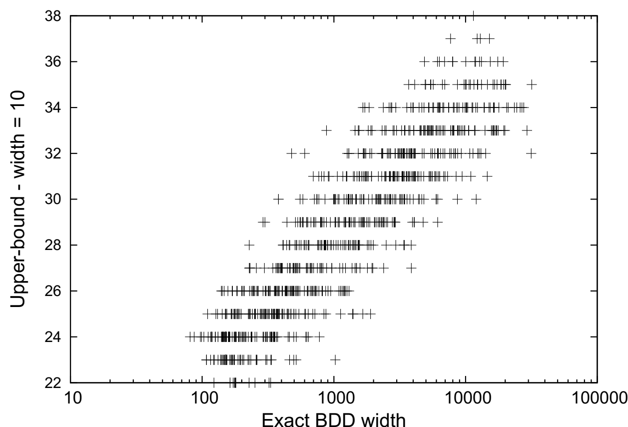


Fig. 7.4 Bound of relaxation BDD vs. exact BDD width for `san200_0.7_1`.

the previous set of experiments. A similar behavior is obtained if the same chart is plotted for other instances.

7.4.3 Relaxation Bounds

We now report the upper bound provided by the relaxation BDD for the original benchmark set, considering all heuristic orderings suggested for maximum widths 100, 500, and 1000. In addition, we generate 100 random orderings generated

uniformly at random, denoted here by RAND, and the bound reported is obtained by taking the average over the 100 generated orderings. The average compilation time for maximum width 100, 500, and 1000 was 0.21, 1.49, and 3.01 seconds, respectively, for the MIN ordering (which was similar to RAND and MPD), while the average time for maximum width 100, 500, and 1000 was 65.01, 318.68, and 659.02, respectively, for the 1LA ordering. For comparison purposes, we have also included the upper bound obtained by considering the IP formulation of the MISP, since this corresponds to a well-known bounding technique for general domains. We ran these instances with CPLEX 12.2 with default settings and took the resulting bound obtained after the root node was computed. We impose a time limit of 60 seconds so that the results were comparable to the MIN ordering with width 1000 since the longest time to create any relaxation BDD with these parameters was for `C.4000.5`, which took 50.42 seconds.

The results are presented in Table 7.3. We report for each instance the optimal or the best known feasible solution and the bounds, where *CPLEX* is the bound obtained by the root node relaxation using CPLEX (the notation 1.00E+75 indicates that a bound was not obtained in the 60-second time limit). By first comparing the results obtained between orderings, we see that the MIN ordering and the general-purpose 1LA heuristic provide the best bounds for most instances. We highlight here that MIN and 1LA were the heuristics that provided the smallest BDD widths for the instances tested in Section 7.4.2. We note that MIN generates BDDs an average of an order of magnitude faster than 1LA.

To compare the obtained bounds with CPLEX, we consider the *relative bound* measure, which is given by (upper bound/optimum). The average relative bound for CPLEX (omitting the instances for which CPLEX was unable to provide a bound) is given by 3.85, while for MIN and 1LA it is given by 2.34 and 2.32, respectively, for a width of 100, and 1.92 and 1.90, respectively, for a width of 1000 (the averages are not significantly different at the 5% level between MIN and 1LA). The average relative ordering for RAND was 5.51 and 4.25 for widths of 100 and 1000, respectively. This indicates that variable orderings are crucial to obtain tighter and relevant bounds, being particularly significant for larger instances when comparing with CPLEX, explaining the smaller average relative bound. We further observe that, since times were very small for the structured heuristics, the bounds obtained here can be improved using the general-purpose bound-improving procedures in [28].

Table 7.3 Benchmark problems relaxations

Maximum Width:		*100*				*500*				*1000*					*1000*
Instance	OPT	MIN	MAX	RAND	1LA	MIN	MAX	RAND	1LA	MIN	MAX	RAND	1LA	CPLEX(1 min.)	MIN
C1000.9.clq	68	261	419	585.42	259	244	394	528.25	241	240	384	506.63	238	**221.78**	240
C125.9.clq	34	46	55	71.68	44	45	52	64.51	42	43	50	61.78	41	**41.2846**	43
C2000.5.clq	16	153	353	368.34	152	121	249	252.27	120	110	218	218	110	1.00E+75	**110**
C2000.9.clq	77	480	829	1170.91	479	447	788	1055.26	447	436	767	1012.4	433	1.00E+75	**436**
C250.9.clq	44	80	107	144.84	78	74	99	130.46	73	72	98	125.21	72	**70.9322**	72
C4000.5.clq	18	281	708	736.31	280	223	497	504.46	223	202	429	435.31	203	1.00E+75	**202**
C500.9.clq	57	142	215	291.48	142	134	203	262.57	133	132	198	251.8	131	**123.956**	132
gen200_p0.9_44.clq	44	62	84	115.69	62	61	79	103.98	59	59	78	99.78	56	**44**	59
gen200_p0.9_55.clq	55	67	88	116.39	65	63	84	104.88	62	61	81	100.57	59	**55**	61
gen400_p0.9_55.clq	55	100	168	233.15	100	99	161	210.21	96	94	156	201.84	94	**55**	94
gen400_p0.9_65.clq	65	112	168	233.63	110	105	161	210.55	105	103	159	202.11	101	**65**	103
gen400_p0.9_75.clq	75	118	170	234.23	118	109	164	211.2	109	108	158	202.73	105	**75**	108
brock200_1.clq	21	42	64	72.12	41	36	54	58.61	36	34	50	54.01	35	38.9817	**34**
brock200_2.clq	12	22	35	35.6	22	17	24	24.68	18	16	22	21.69	16	22.3764	**16**
brock200_3.clq	15	28	48	48.87	29	24	36	36.22	25	23	33	32.39	23	28.3765	**23**
brock200_4.clq	17	32	53	56.61	32	29	42	43.32	27	26	37	39.12	25	31.5437	**26**
brock400_1.clq	27	72	127	145.81	71	63	108	118.75	63	60	102	109.32	61	67.2201	**60**
brock400_2.clq	29	75	128	147.35	72	63	107	119.47	61	61	101	110.16	60	67.9351	**61**
brock400_3.clq	31	72	127	146.19	73	64	109	118.63	64	60	102	109.12	60	67.4939	**60**
brock400_4.clq	33	70	129	146.43	71	63	110	119.54	63	63	106	109.59	61	67.3132	**63**
brock800_1.clq	23	99	204	222.01	100	85	160	168.39	86	79	145	151.21	78	136.103	**79**
brock800_2.clq	24	101	201	224.38	100	86	162	170.65	85	79	145	153.29	79	136.538	**79**
brock800_3.clq	25	101	203	222.61	100	84	164	169.05	84	81	149	151.31	79	130.832	**81**
brock800_4.clq	26	101	205	223.41	100	84	161	169.81	84	80	145	152.66	78	132.696	**80**
c-fat200-1.clq	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
c-fat200-2.clq	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
c-fat200-5.clq	58	58	58	58	58	58	58	58	58	58	58	58	58	61.6953	**58**
c-fat500-1.clq	14	14	15	16.62	14	14	14	14	14	14	14	14	14	230.513	**14**
c-fat500-10.clq	126	126	126	126	126	126	126	126	126	126	126	126	126	246	**126**
c-fat500-2.clq	26	26	26	26	26	26	26	26	26	26	26	26	26	240	**26**
c-fat500-5.clq	64	64	64	64	64	64	64	64	64	64	64	64	64	244.5	**64**
hamming10-2.clq	512	512	512	892.69	515	512	512	871.68	512	512	512	862.99	512	512	512
hamming10-4.clq	40	106	91	456.63	105	96	76	385.13	93	79	72	359.76	79	206.047	**79**
hamming6-2.clq	32	32	32	37.01	32	32	32	34.03	32	32	32	33.28	32	32	32
hamming6-4.clq	4	4	4	5.98	4	4	4	4	4	4	4	4	4	5.33333	**4**
hamming8-2.clq	128	128	128	194.42	128	128	128	184.51	128	128	128	180.71	128	128	128
hamming8-4.clq	16	20	21	62.23	19	18	18	45.66	18	17	17	40.56	17	**16**	17
johnson16-2-4.clq	8	11	11	38.75	11	9	9	29.24	9	8	8	25.64	8	8	8
johnson32-2-4.clq	16	40	35	250.07	42	38	29	215.06	39	35	25	202.36	40	**16**	35
johnson8-2-4.clq	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
johnson8-4-4.clq	14	14	15	24.57	14	14	14	19.82	14	14	14	18.54	14	14	14
keller4.clq	11	19	22	43.38	18	16	17	31.24	16	15	16	27.54	15	**14.75**	15
keller5.clq	27	58	98	280.74	59	56	77	225.75	55	48	72	207.08	49	**32.875**	48
keller6.clq	59	171	417	1503.26	174	142	332	1277.98	144	123	307	1197.76	125	1.00E+75	**123**
MANN_a27.clq	126	142	138	327.2	135	140	137	318.93	137	139	137	315.25	136	**133.331**	139
MANN_a45.clq	345	371	365	954.51	366	368	362	942.45	363	368	362	937.06	365	**357.162**	368
MANN_a81.clq	1100	1154	1143	3186.21	1141	1150	1143	3166.06	1143	1148	1143	3158.78	1141	**1131.82**	1148
MANN_a9.clq	16	18	18	27.21	17	16	16	23.9	16	16	16	22.88	16	17	**16**
p_hat1000-1.clq	10	47	86	88.73	48	35	52	52.71	36	31	43	43.37	31	413.5	**31**
p_hat1000-2.clq	46	130	210	225.57	129	116	171	178.1	112	112	159	163.47	108	376.5	**112**
p_hat1000-3.clq	68	202	324	383.76	197	187	286	322.62	179	179	272	302.07	175	245.674	**179**
p_hat1500-1.clq	12	68	136	139.02	68	51	83	83.08	51	46	69	68.33	45	1.00E+75	**46**
p_hat1500-2.clq	65	199	344	357.01	193	176	285	286.03	174	168	267	263.95	163	1.00E+75	**168**
p_hat1500-3.clq	94	298	511	594.04	296	277	452	502.22	270	272	433	470.91	266	1.00E+75	**272**
p_hat300-1.clq	8	17	27	26.05	18	14	16	15.89	14	12	13	13.39	12	18.2278	**12**
p_hat300-2.clq	25	48	64	66.46	45	42	51	52.29	40	40	48	47.83	39	**35.2878**	40
p_hat300-3.clq	36	70	99	114.66	67	65	89	95.93	61	62	84	89.86	60	**55.2598**	62
p_hat500-1.clq	9	28	45	45.33	27	21	28	27.3	21	18	23	22.7	19	158	**18**
p_hat500-2.clq	36	77	112	116.55	72	69	92	92.8	64	66	84	85.54	63	160.25	**66**
p_hat500-3.clq	50	111	172	195.67	109	106	155	165.35	102	104	147	154.88	99	**90.7331**	104
p_hat700-1.clq	11	36	62	63.27	36	27	39	37.83	27	24	31	31.33	24	272.5	**24**
p_hat700-2.clq	44	101	155	163.03	99	90	128	130.39	88	85	118	120.19	83	272.5	**85**
p_hat700-3.clq	62	153	234	272.83	147	142	208	230.14	141	137	198	215.93	134	160.333	**137**
san1000.clq	15	28	184	202.02	26	21	101	104.09	19	19	78	79.84	19	462.5	**19**
san200_0.7_1.clq	30	32	66	73.67	31	30	57	60.3	30	30	52	55.37	30	30	30
san200_0.7_2.clq	18	23	58	71.76	21	20	48	56.2	20	19	46	50.23	18	**18**	19
san200_0.9_1.clq	70	71	86	118.89	70	70	82	108.56	70	70	81	105.13	70	70	70
san200_0.9_2.clq	60	68	86	116.48	64	64	83	105.39	60	60	81	101.05	60	60	60
san200_0.9_3.clq	44	57	84	115	54	55	78	103.23	53	51	77	99	52	**44**	51
san400_0.5_1.clq	13	17	66	69.02	18	14	35	35.6	14	13	28	28.31	13	13	13
san400_0.7_1.clq	40	50	142	160.35	51	46	127	136.08	43	42	119	126.86	41	**40**	42
san400_0.7_2.clq	30	44	129	147.55	45	38	108	119.96	39	37	103	109.84	35	**30**	37
san400_0.7_3.clq	22	36	118	137.72	38	29	98	108.29	31	29	91	97.98	29	**22**	29
san400_0.9_1.clq	100	117	175	236.22	118	109	169	214.05	108	108	164	205.73	108	**100**	108
sanr200_0.7.clq	18	34	58	63	36	31	46	49.56	32	30	44	45.18	29	34.5339	**30**
sanr200_0.9.clq	42	67	86	114.78	66	63	83	103.25	60	61	80	98.89	61	**59.5252**	61
sanr400_0.5.clq	13	40	70	73.32	39	33	50	50.5	31	29	45	43.73	29	43.1544	**29**
sanr400_0.7.clq	21	64	115	128.44	64	55	96	101.06	54	52	89	91.69	52	62.078	**52**

Chapter 8
Recursive Modeling

Abstract This chapter focuses on the type of recursive modeling that is required for solution by decision diagrams. It presents a formal development that highlights how solution by decision diagrams differs from traditional enumeration of the state space. It illustrates the versatility of recursive modeling with examples: single-facility scheduling, scheduling with sequence-dependent setup times, and minimum bandwidth problems. It shows how to represent state-dependent costs with canonical arc costs in a decision diagram, a technique that can sometimes greatly simplify the recursion, as illustrated by a textbook inventory management problem. It concludes with an extension to nonserial recursive modeling and nonserial decision diagrams.

8.1 Introduction

The optimization and constraint solving communities have developed two primary modeling styles, one based on constraints and one on recursive formulations. Constraint-based modeling is the norm in mathematical programming, where constraints almost invariably take the form of inequalities or equations, as well as in constraint programming, which draws from a collection of high-level global constraints. Recursive modeling, on the other hand, characterizes dynamic programming and Markov decision processes.

Both modeling paradigms have seen countless successful applications, but it is hard to deny that constraint-based modeling is the dominant one. Recursive modeling is hampered by two perennial weaknesses: it frequently results in state spaces that grow exponentially (the "curse of dimensionality"), and there are no

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_8

general-purpose solvers for recursive models, as there are for mathematical programming and constraint programming models. An extensive literature shows how to overcome the curse of dimensionality in many applications, using state space relaxation, approximate dynamic programming, and the like. Yet these require highly tailored solution algorithms, and many other recursive formulations remain intractable. As a result, the major inherent advantage of recursive modeling too often goes unexploited: its ability to model a vast range of feasible sets and objective functions, with no need for linear, convex, or closed-form expressions.

Solution methods based on decision diagrams can accommodate both types of modeling, as illustrated throughout this book. However, decision diagrams have a special affinity to recursive modeling due to their close relationship with state transition graphs in deterministic dynamic programming. Furthermore, they offer the prospect of addressing the two weaknesses of recursive modeling in a novel fashion. The use of relaxed decision diagrams allows recursive models to be solved by branch-and-bound methods rather than by enumerating the state space, as described in Chapter 6. This, in turn, may allow the development of general-purpose branch-and-bound solvers that are analogous to mixed integer programming solvers. Decision diagrams may therefore help to unlock the unrealized potential of recursive modeling.

Recursive modeling may seem restrictive at first, because it requires that the entire problem be formulated in a sequential, Markovian fashion. Each stage of the recursion can depend only on the previous stage. This contrasts with a constraint-based formulation, in which constraints can be added at will, with no need for a particular overall structure. The only requirement is that the individual constraints be in recognizable form, as for example linear inequalities.

However, once an overall recursive structure is identified, recursive modeling provides enormous flexibility. Any objective function or feasible set that can be expressed in terms of the current state and control can be modeled, either in closed form or by subroutine call. A wide range of problems naturally have this structure, and these frequently have no convenient mathematical programming formulation. Many other problems can, with a little ingenuity, be put in the required form.

A particular strength of recursive models is that *any possible* objective function over finite domains can be modeled with state-dependent costs, which correspond to arc costs on the corresponding decision diagram. In fact, a single objective function can be realized with several different sets of arc costs, one of which can be described as canonical. A straightforward conversion of arc costs to canonical costs can result

in a smaller reduced diagram. The perspective afforded by decision diagrams can therefore lead to a simpler and more easily solved recursive model. This is illustrated below with an elementary inventory management problem.

Finally, recursive models can be further extended to nonserial dynamic programming models, in which the linear layout of a conventional dynamic programming model becomes a directed acyclic graph. Decision diagrams can be similarly extended to nonserial decision diagrams.

8.2 General Form of a Recursive Model

We first recall the general form of a recursive model for decision diagrams that was set out in Chapters 3 and 4. The model consists of an exact formulation and a node merger rule for creating a relaxed decision diagram.

The exact formulation consists of control variables (or controls, for short), a state space, and transition functions. The controls are $x_1,\ldots,x_n$ and have finite domains $D(x_1),\ldots,D(x_n)$, respectively. The state space S is the union of sets $S_1,\ldots,S_{n+1}$ corresponding to stages of the recursion. The initial set S_1 contains only an initial state $\hat{r}$, and the final set S_{n+1} contains one or more terminal states. In addition, each S_i contains an infeasible state $\hat{0}$. For each stage $i=1,\ldots,n$ there is a state transition function $t_i : S_i \times D(x_i) \to S_{i+1}$, where $t_i(s^i,x_i)$ specifies the result of applying control x_j in state s^i. The infeasible state always transitions to itself, so that $t_i(\hat{0},x_i)=\hat{0}$ for all $x_i \in D(x_i)$. There are also immediate cost functions $h_i : S_i \times D(x_i) \to \mathbb{R}$ for $i=1,\ldots,n$, where $h_i(s^i,x_i)$ is the immediate cost of applying control x_i in state s^i. The optimization problem is to identify controls that minimize

$$F(x_1,\ldots,x_n)=\sum_{i=1}^{n} h_i(s^i,x_i) \tag{8.1}$$

subject to

$$\begin{aligned} & s^{i+1}=t_i(s^i,x_i),\ x_i \in D(x_i),\ \ i=1,\ldots,n, \\ & s^i \in S_i,\ \ i=1,\ldots,n+1. \end{aligned} \tag{8.2}$$

The summation in the objective function (8.1) can be replaced by another operator, such as a product, maximum, or minimum.

The problem is classically solved by backward induction:

$$g_i(s^i) = \min_{x_i \in D(x_i)} \left\{ h_i(s^i, x_i) + g_{i+1}(t_i(s^i, x_i)) \right\}, \text{ all } s^i \in S_i,\ i = 1, \ldots, n \tag{8.3}$$

with boundary condition

$$g_{n+1}(s^{n+1}) = \begin{cases} 0 & \text{if } s^{n+1} \in S_{n+1} \setminus \{\hat{0}\} \\ \infty & \text{if } s^{n+1} = \hat{0} \end{cases},$$

where $g_i(s^i)$ is the *cost-to-go* at state s^i. The optimal value is $g_1(\hat{r})$. The sum in (8.3) can again be replaced by another operator. Backward induction requires, of course, that the elements of the state space S be enumerated, frequently a task of exponential complexity. Optimal solutions are recovered in a forward pass by letting $X_i(s^i)$ be the set of controls $x_i \in D(x_i)$ that achieve the minimum in (8.3) for state s^i. Then an optimal solution is any sequence of controls $(\bar{x}_1, \ldots, \bar{x}_n)$ such that $\bar{x}_i \in X_i(\bar{s}^i)$ for $i = 1, \ldots, n$, $\bar{s}^1 = \hat{r}$, and $\bar{s}^{i+1} = t_i(\bar{s}^i, \bar{x}_i)$ for $i = 1, \ldots, n-1$.

The *state transition graph* for a recursive model is defined recursively. The initial state $\hat{r}$ corresponds to a node of the graph, and for every state s^i that corresponds to a node of the graph and every $x_i \in D(x_i)$, there is an arc from that node to a node corresponding to state $t_i(s^i, x_i)$. The arc has length equal to the immediate cost $h_i(s^i, x_i)$, and a shortest path from $\hat{r}$ to a terminal state corresponds to an optimal solution.

The state transition graph can be regarded as a decision diagram, after some minor adjustments: remove nodes corresponding to the infeasible state, and add an arc from each terminal state to a terminal node. The result is a decision diagram in which each layer (except the terminal layer) corresponds to a stage of the recursion.

A relaxed decision diagram is created by a relaxation scheme $(\oplus, \Gamma)$ that merges states associated with nodes of the original decision diagram. The operator $\oplus$ maps a set M of states corresponding to nodes on layer i to a single state $\oplus(M)$. The function Γ maps the immediate cost $v = h_{i-1}(s^{i-1}, x_{i-1})$ of any control x_{i-1} that leads to state $s^i \in M$ to a possibly altered cost $\Gamma_M(s^i, v)$. This results in modified transition and cost functions t'_{i-1}, h'_{i-1} that are identical to the original functions except that they reflect the redirected arcs and altered costs:

$$\left.\begin{aligned} t'_{i-1}(s^{i-1}, x_{i-1}) &= \oplus(M) \\ h'_{i-1}(s^{i-1}, x_{i-1}) &= \Gamma_M(s^{i-1}, t_{i-1}(s^{i-1}, x_{i-1})) \end{aligned}\right\} \text{whenever } h_{i-1}(s^{i-1}, x_{i-1}) \in M,$$

The relaxation scheme is valid if every feasible solution of the original recursion is feasible in the modified recursion and has no greater cost. For this it suffices that

$$t_i(s^i, x_i) \neq \hat{0} \;\Rightarrow\; t_i(\oplus(M), x_i) \neq \hat{0}, \;\; \text{for all } s^i \in M,\; x_i \in D(x_i),$$
$$\Gamma_M(s^{i-1}, v) \leq v, \;\; \text{for all } s^{i-1} \in S_{i-1},\; v \in \mathbb{R}.$$

It is frequently convenient to modify the exact recursion somewhat to make it amenable to state merger. This is illustrated in the next section.

8.3 Examples

Problems with a natural recursive structure frequently have no convenient constraint-based formulation. Sequencing and minimum bandwidth problems provide good examples.

8.3.1 Single-Machine Scheduling

In a simple single-machine scheduling problem, each job j has processing time p_j and deadline d_j. A typical objective is to minimize total tardiness.

At least six integer programming models have been proposed for the problem, surveyed by [12]. However, there is a natural recursive model for single-machine scheduling. The model also accommodates a variety of side constraints that have no practical integer programming formulation. The control x_i is the i-th job processed. The state $s^i = (J_i, f_i)$ at stage i consists of the set J_i of jobs so far processed and the finish time f_i of the last job processed. The initial state is $(\emptyset, 0)$. Because the next job to be processed cannot be in J_i, selecting control $x_i = j$ effects the transition

$$t_i((J_i, f_i), j) = \begin{cases} (J_i \cup \{j\},\; f_i + p_j), & \text{if } j \notin J_i \\ \hat{0}, & \text{if } j \in J_i \end{cases}.$$

The immediate cost $h_i(J_i, j)$ is the tardiness $(f_i + p_j - d_j)^+$ of job j, where the notation α^+ indicates $\max\{0, \alpha\}$. A simple relaxation scheme merges a subset M of states to obtain

$$\oplus(M) = \Big(\bigcap_{(J_i, f_i) \in M} J_i,\; \min_{(J_i, f_i) \in M} \{f_i\} \Big),$$

where f_i is now interpreted as the earliest possible finish time of the most recent job. The immediate costs are unchanged.

This recursive model readily accommodates any side constraint or objective function that can be defined in terms of (J_i, f_i) and x_j. For example, release times r_j for the jobs are accommodated by the slightly different transition function

$$t_i((J_i, f_i), j) = \begin{cases} (J_i \cup \{j\},\ \max\{r_j, f_i\} + p_j), & \text{if } j \notin J_i \\ \hat{0}, & \text{if } j \in J_i \end{cases}$$

and immediate cost $h_i((J_i, f_i), j) = (\max\{r_j, f_i\} + p_j - d_j)^+$. One can shut down the machine for maintenance in the interval $[a, b]$ by using the transition function

$$t_i((J_i, f_i), j) = \begin{cases} (J_i \cup \{j\},\ f_i + p_j + b - a), & \text{if } j \notin J_i \text{ and } f_i \in [a - p_j, b) \\ (J_i \cup \{j\},\ f_i + p_j), & \text{if } j \notin J_i \text{ and } f_i \notin [a - p_j, b) \\ \hat{0}, & \text{if } j \in J_i \end{cases}$$

and immediate cost

$$h_i((J_i, f_i), j) = \begin{cases} (f_i + p_j + b - a - d_j)^+, & \text{if } f_i \in [a - p_j, b) \\ (f_i + p_j - d_j)^+, & \text{otherwise.} \end{cases}$$

In addition, processing job j may require that certain components have already been fabricated in the processing of previous jobs. We simply set $t_i(J_i, j) = \hat{0}$ when the jobs in J_i do not yield the necessary components. Such side constraints actually make the problem easier by simplifying the decision diagram.

A wide variety of objective functions are also possible. For example, the cost of processing job j may depend on which jobs have already been processed, perhaps again due to common components. We can let the cost associated with control $x_j = j$ be any desired function $c_j(J_i)$, perhaps evaluated by a table lookup. Or the cost could be an arbitrary function $c_j((f_i + p_j - d_j)^+)$ of tardiness, such as a step function, or a function of both J_i and f_i.

8.3.2 Sequence-Dependent Setup Times

The single-machine model can be modified to allow for sequence-dependent setup times. The transition time from job ℓ to job j is now $p_{\ell j}$. An exact recursive model can be formulated by introducing a third state variable ℓ_i to indicate the last job

processed. This yields the transition function

$$t_i((J_i, \ell_i, f_i), j) = \begin{cases} (J_i \cup \{j\}, j, f_i + p_{\ell_i j}), & \text{if } j \notin J_i \\ \hat{0}, & \text{if } j \in J_i \end{cases} \tag{8.4}$$

with immediate cost $h_i((J_i, \ell_i, f_i), j) = (f_i + p_{\ell_i j} - d_j)^+$.

To create a relaxation scheme, we must allow for the possibility that states with different values of ℓ_i will be merged. The pair (ℓ_i, f_i) of state variables therefore becomes a set L_i of pairs, any one of which could represent the last job processed. The transition function is now

$$t_i((J_i, L_i), j) = \begin{cases} \left(J_i \cup \{j\}, \left\{\left(j, \min\limits_{(\ell_i, f_i) \in L_i} \{f_i + p_{\ell_i j}\}\right)\right\}\right), & \text{if } j \notin J_i \\ \hat{0}, & \text{if } j \in J_i \end{cases} \tag{8.5}$$

with immediate cost

$$h_i((J_i, L_i), j) = \left(\min_{(\ell_i, f_i) \in L_i} \{f_i + p_{\ell_i j}\} - d_j\right)^+.$$

Note that the finish time of job j is based on the previous job ℓ_i that results in the earliest finish time for job j. This ensures a valid relaxation. States are merged by taking the intersection of the sets J_i, as before, and the union of the sets L_i:

$$\oplus(M) = \left(\bigcap_{(J_i, L_i) \in M} J_i, \bigcup_{(J_i, L_i) \in M} L_i\right). \tag{8.6}$$

As in the previous section, the model accommodates any side constraint that can be defined in terms of the current state and control.

The famous traveling salesman problem results when deadlines are removed and the objective is to minimize total travel time. The transition function (8.4) for the exact recursion simplifies to

$$t_i((J_i, \ell_i), j) = \begin{cases} (J_i \cup \{j\}, j), & \text{if } j \notin J_i \\ \hat{0}, & \text{if } j \in J_i \end{cases}$$

with immediate cost $h_i((J_i, \ell_i), j) = p_{\ell_i j}$. This is the classical dynamic programming model for the problem, which is generally impractical to solve by backward induction because of the exponential state space. However, the relaxation scheme given above allows a recursive model to be solved by branch and bound. The set L_i of

pairs (ℓ_i, f_i) becomes a set of jobs ℓ_i that could be the last job processed, and the transition function (8.5) simplifies to

$$t_i((J_i, L_i), j) = \begin{cases} (J_i \cup \{j\}, \{j\}), & \text{if } j \notin J_i \\ \hat{0}, & \text{if } j \in J_i \end{cases}$$

with immediate cost

$$h_i((J_i, L_i), j) = \min_{\ell_i \in L_i} \{p_{\ell_i j}\}.$$

The node merger rule (8.6) is unchanged.

A wide variety of integer programming models have been proposed for the traveling salesman problem (see [121] for a survey). The only model that has proved useful in practice relies on subtour elimination constraints. Because there are exponentially many such constraints, this model is not actually used to formulate the problem for a solver. Rather, specialized solvers are built that generate subtour elimination constraints and other valid inequalities as needed. By contrast, there is a compact and natural recursive model for the problem, described above, that can serve as input to a branch-and-bound solver based on decision diagrams. One can also add side constraints that are difficult to formulate in an integer programming framework. Chapter 11 discusses the use of decision diagrams in the context of single-machine scheduling in more detail.

8.3.3 Minimum Bandwidth

Another modeling illustration is provided by the minimum bandwidth and linear arrangement problems, both of which are defined on an undirected graph $G = (V, E)$ of n vertices. The minimum bandwidth problem assigns a distinct vertex $x_i \in V$ to each position $i = 1, \ldots, n$ so as to minimize

$$\max_{(x_i, x_j) \in E} \{|i - j|\},$$

where $|i - j|$ is the *length* of edge (x_i, x_j) in the ordering $x_1, \ldots, x_n$. The linear arrangement problem differs only in its objective function, which is to minimize

$$\sum_{(x_i, x_j) \in E} |i - j|.$$

It is not obvious how to write integer programming models for these problems, and [40] observes that no useful integer programming models are known.

A recursive model, however, can be formulated as follows. A state at stage i is a tuple $s^i = (s^i_1,\ldots,s^i_i)$, where s^i_ℓ is interpreted as the vertex assigned to position ℓ. This is a brute-force model in the sense that there is only one feasible path from the root to each state, namely the path that sets $(x_1,\ldots,x_i) = (s^i_1,\ldots,s^i_i)$. However, the model will become more interesting when relaxed. The state transition function is

$$t_i(s^i,j) = \begin{cases} (s^i_1,\ldots,s^i_i,j), & \text{if } j \notin \{s^i_1,\ldots,s^i_i\} \\ \hat{0}, & \text{otherwise.} \end{cases}$$

Since vertex j is placed in position $i+1$, the immediate cost is

$$h_i(s^i,j) = \max_{\substack{k=1,\ldots,i \\ (s^i_k,j)\in E}} \{i+1-k\}$$

for the minimum bandwidth problem and

$$h_i(s^i,j) = \sum_{\substack{k=1,\ldots,i \\ (s^i_k,j)\in E}} (i+1-s^i_k)$$

for the linear arrangement problem.

To formulate a relaxation, one can modify the recursion so that the state is a tuple $S^i = (S^i_1,\ldots,S^i_i)$, where S^i_ℓ is interpreted as the set of values that have been assigned to x_ℓ along paths to the state S^i. To state the transition function, one must determine what values x_{i+1} can take, consistent with the current state. This can be determined by maintaining domain consistency for ALLDIFFERENT$(x_1,\ldots,x_i,x_{i+1})$ given domains $x_\ell \in S^i_\ell$ for $\ell = 1,\ldots,i$ and $x_{i+1} \in \{1,\ldots,n\}$.[1] If S^*_{i+1} is the filtered domain for x_{i+1} that results, then the transition function is

$$t_i(S^i,j) = \begin{cases} (S^i_1,\ldots,S^i_i,\{j\}), & \text{if } j \in S^*_{i+1} \\ \hat{0}, & \text{otherwise.} \end{cases}$$

To compute the immediate cost, we let the *lower-bound length* $\mathrm{LB}_{jk}(S^i)$ of an edge (j,k) be the minimum length of (j,k) over all positions that can be assigned to vertex k given the current state S^i. Thus the lower-bound length of (j,k) is

[1] The concepts of domain filtering and domain consistency in constraint programming are discussed in Chapter 9.

$$\mathrm{LB}_{jk}(S^i) = \min_{\substack{\ell \\ k \in S^i_\ell}} \{|i+1-\ell|\}.$$

The immediate cost for the minimum bandwidth problem is now the maximum lower-bound length of edges incident to vertex j and a vertex in $S^i_1 \cup \cdots \cup S^i_i$:

$$h_i(S^i, j) = \max_{k \in S^i_1 \cup \cdots \cup S^i_i} \{\mathrm{LB}_{jk}(S^i)\}.$$

The immediate cost for the linear arrangement problem is similar:

$$h_i(S^i, j) = \sum_{k \in S^i_1 \cup \cdots \cup S^i_i} \mathrm{LB}_{jk}(S^i).$$

The state merger rule takes a componentwise union of the states:

$$\oplus(M) = \Big(\bigcup_{S^i \in M} S^i_1, \; \ldots, \; \bigcup_{S^i \in M} S^i_i \Big).$$

8.4 State-Dependent Costs

One advantage of recursive modeling is its ability to accommodate state-dependent costs in the objective function (8.1). That is, the cost incurred in each state is a function of the current state as well as the control. This not only provides great flexibility in modeling the objective function, but in problems with finite variable domains (as we are considering), it allows one to model *every possible* objective function.

This insight and others become evident when one takes a perspective based on decision diagrams. State-dependent costs allow one to put a different cost on every arc of the corresponding decision diagram. This not only makes it possible to model every possible objective function, but there are multiple ways to assign costs to arcs so as to formulate any given objective function.

It is convenient to refer to a decision diagram with costs assigned to its arcs as a *weighted* decision diagram. Different weighted diagrams can represent the same optimization problem, just as different unweighted diagrams can represent the same feasible set. Furthermore, just as there is a unique *reduced* unweighted diagram for a given feasibility problem (for a fixed variable ordering), there is a unique reduced weighted diagram for a given optimization problem, provided the costs are assigned

Table 8.1 (a) A small set covering problem. The dots indicate which elements belong to each set i. (b) A nonseparable cost function for the problem. Values are shown only for feasible x.

(a)

	Set i 1	2	3	4
A	•	•		
B	•		•	•
C		•	•	
D		•		•

(b)

x	$F(x)$
(0,1,0,1)	6
(0,1,1,0)	7
(0,1,1,1)	8
(1,0,1,1)	5
(1,1,0,0)	6
(1,1,0,1)	8
(1,1,1,0)	7
(1,1,1,1)	9

to arcs in a certain canonical fashion. In fact, a canonical cost assignment may allow one to simplify the diagram substantially, resulting in a simpler recursive model that can be solved more rapidly. This will be illustrated for a textbook inventory management problem.

8.4.1 Canonical Arc Costs

The properties of state-dependent costs can be illustrated by an example, taken from [97]. Consider the small set covering problem of Table 8.1(a). The goal is to choose a subcollection of the four sets whose union contains elements A, B, C, and D. In a natural recursive formulation of the problem, binary control variable x_i takes the value 1 when set i is included in the subcollection, and the state is the set S_i of sets so far included. The transition function is

$$t_i(S_i, x_i) = \begin{cases} S_i \cup \{i\}, & \text{if } x_i = 1 \\ S_i, & \text{if } x_i = 0 \end{cases}.$$

A reduced decision diagram for the problem appears in Fig. 8.1(a).

The simplest type of objective function is a separable one, meaning that the objective function (8.1) has the form $F(x) = \sum_i h_i(x_i)$. For example, one might assign weight w_i to each vertex i and seek a minimum weight set cover, in which case the objective function sets $h_i(1) = w_i$ and $h_i(0) = 0$. A separable objective function results in a *separable weighted diagram*, meaning that, in any given layer, the arc cost associated with a given control is always the same. Figure 8.1(b) shows the separable decision diagram that results from weights $(w_1, \ldots, w_4) = (3, 5, 4, 6)$.

While separable cost functions are the easiest to model, a weighted decision diagram (and by implication, a recursive model) can accommodate an arbitrary objective function, separable or nonseparable. Consider, for example, the nonseparable cost function shown in Table 8.1(b). This and any other cost function can be represented in a branching tree by placing the cost of every solution on the arc that leads to the corresponding leaf node, and a cost of zero on all other arcs, as shown in Fig. 8.2.

This search tree becomes a weighted decision diagram if the leaf nodes are superimposed to form a terminal node. Furthermore, the decision diagram can be reduced. The reduction proceeds as for an unweighted decision diagram, namely by superimposing isomorphic subdiagrams rooted at nodes in the same layer. A subdiagram rooted at a given node is the portion of the diagram that contains all paths between that node and the terminal node. In a weighted decision diagram, subdiagrams are isomorphic only when corresponding arcs reflect the same cost as well as the same control. This limits the amount of reduction that is possible.

However, it is frequently possible to achieve greater reduction when the arc costs are *canonical*. An assignment of arc costs to a tree or decision diagram is canonical if, for every layer $L_i \geq 2$, and for every node on that layer, the smallest arc cost

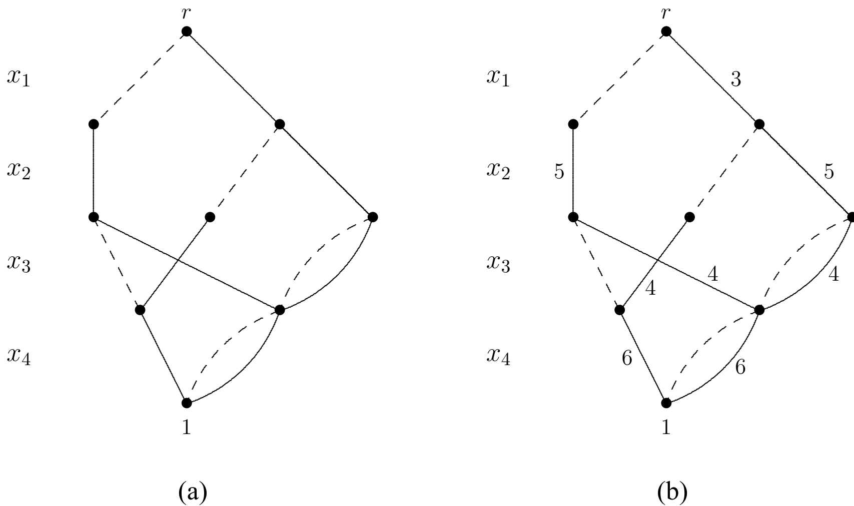


Fig. 8.1 (a) Decision diagram for the set covering problem in Table 8.1(a). Dashed arcs correspond to setting $x_i = 0$, and solid arcs to setting $x_i = 1$. (b) Decision diagram showing arc costs for a separable objective function. Unlabeled arcs have zero cost.

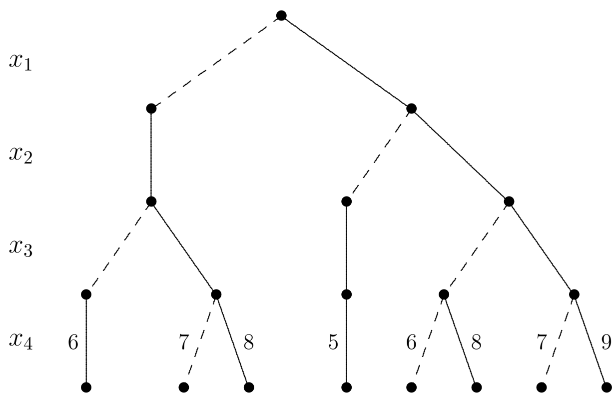


Fig. 8.2 Branching tree for the set covering problem in Table 8.1(a). Only feasible leaf nodes are shown.

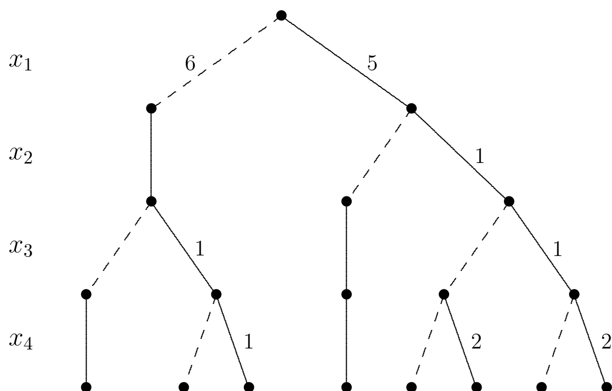


Fig. 8.3 Branching tree with canonical arc costs. Unlabeled arcs have zero cost.

leaving that node is a predefined value α_i. In the simplest case $\alpha_i = 0$, but in some applications it is convenient to allow other values.

A simple algorithm converts any set of arc costs to canonical costs. For each layer L_i, $i = n, n-1, \ldots, 2$, do the following: for each node u in layer L_i, add $\alpha_i - c_{\min}$ to the cost on each arc leaving u, where $c_{\min}$ is the minimum cost on arcs leaving u, and add $c_{\min} - \alpha_i$ to each arc entering u. For example, the costs on the tree of Fig. 8.2 become the canonical costs shown in Fig. 8.3 if each $\alpha_i = 0$. This tree can, in turn, be reduced to the decision diagram in Fig. 8.4(a). Note that this reduced diagram is slightly larger than the reduced unweighted diagram in Fig. 8.1(a), which is to be expected since costs must be matched before subdiagrams are superimposed.

The arc costs of Fig. 8.4(a) represent a state-dependent objective function (8.1). For example, the state at the leftmost node in layer 4 is $S_4 = \{2,3\}$, and the

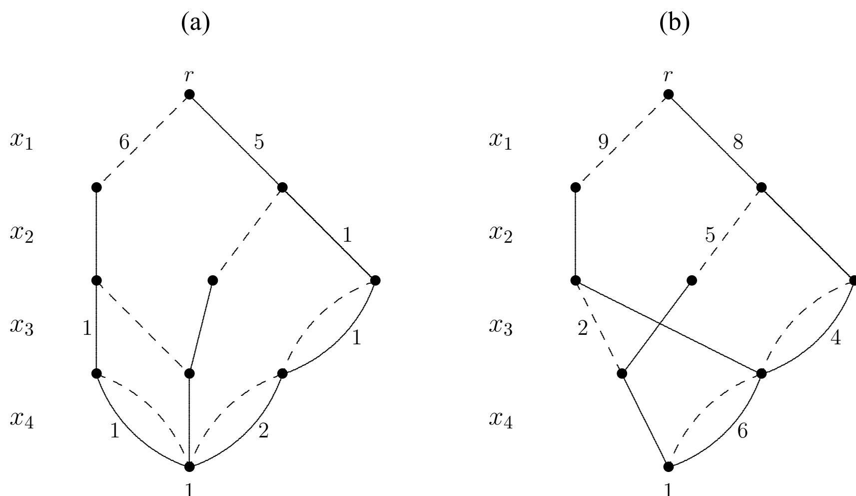


Fig. 8.4 (a) Weighted decision diagram with canonical arc costs for a nonseparable objective function. (b) Canonical arc costs for a separable objective function. Unlabeled arcs have zero cost.

immediate costs for this state are the costs on the arcs leaving the node, namely $h_4(S_4,1) = 1$ and $h_4(S_4,0) = 0$. In general, any possible objective function can be represented by state-dependent costs because it can be represented by a weighted decision diagram.

Furthermore, two uniqueness results can be proved [97]. There is a unique canonical assignment of arc costs representing a given objective function, once the offsets α_i are fixed. In addition, the classical uniqueness theorem for reduced, ordered diagrams [37] can be extended to weighted diagrams.

Theorem 8.1. *Any discrete optimization problem (8.1)–(8.2) is represented by a unique reduced weighted decision diagram with canonical arc costs, for a given variable ordering and fixed offsets* α_i, $i = 2,\ldots,n$.

Interestingly, conversion of costs on a separable diagram to canonical costs can result in a nonseparable diagram. For example, the separable costs of Fig. 8.1(b) result in the nonseparable diagram of Fig. 8.4 when converted to canonical costs. Nonetheless, it is shown in [97] that converting a separable diagram to canonical arc costs has no effect on the size or shape of the *reduced* diagram. So there is never a penalty for converting separable costs to canonical costs.

Theorem 8.2. *A weighted decision diagram that is reduced when costs are ignored remains reduced when its arc costs are converted to canonical costs.*

8.4.2 Example: Inventory Management

A textbook example in inventory management illustrates how converting costs to canonical costs can lead to substantial reduction of the weighted decision diagram. This simplifies the recursion and allows faster solution.

The objective is to adjust production quantities and inventory levels to meet demand over n periods while minimizing production and holding costs. The control is the production quantity x_i in period i, and the state is the stock on hand s_i at the beginning of period i. Costs include a unit production cost c_i and unit holding cost of h_i in period i. The demand in period i is d_i, and the warehouse has capacity m in each period. The state transition function is

$$g_i(s_i,x_i) = \begin{cases} s_i + x_i - d_i, & \text{if } 0 \le s_i + x_i - d_i \le m \\ \hat{0}, & \text{otherwise} \end{cases}$$

with immediate cost $h_i(s_i,x_i) = c_i x_i + h_i s_i$. To simplify exposition, it is assumed that the production level x_i can be negative, and that this corresponds to selling off inventory at a price equal to the current production cost.

The decision diagram has the form shown in Fig. 8.5(a). Note that the set of arcs leaving any node is essentially identical to the set of arcs leaving any other node in the same stage. The controls x_i and the costs are different, but the controls can be equalized by a simple change of variable, and the costs can be equalized by transforming them to canonical costs. This will allow the diagram to be reduced substantially.

To equalize the controls, let the control x_i' be the stock level at the beginning of the next stage, so that $x_i' = s_i + x_i - d_i$. Then the controls leaving any node are $x_i' = 0,\ldots,m$. The transition function becomes simply

$$g_i(s_i,x_i') = \begin{cases} x_i', & \text{if } 0 \le x_i' \le m \\ \hat{0}, & \text{otherwise} \end{cases}.$$

with immediate cost

$$h_i(s_i,x_i') = c_i(x_i' - s_i + d_i) + h_i s_i$$

To transform the costs to canonical costs, subtract $h_i s_i + (m - s_i)c_i$ from the cost on each arc (s_i,s_{i+1}), and add this amount to each arc coming into s_i. Then for any period i, the arcs leaving any given node s_i have the same set of costs. Specifically, realizing that x_i' represents s_{i+1}, arc (s_i,s_{i+1}) has cost

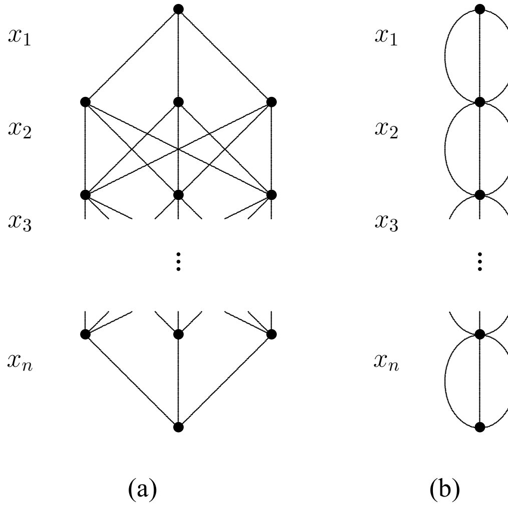


Fig. 8.5 (a) State transition graph for a production and inventory management problem. (b) Reduced state transition graph after converting costs to canonical costs.

$$\bar{c}_i(s_{i+1}) = (d_i + s_{i+1} - m)c_i + s_{i+1}h_{i+1} + (m - s_{i+1})c_{i+1}$$

and so depends only on the next state s_{i+1}. These costs are canonical for the offsets

$$\alpha_i = \min_{s_{i+1}\in\{0,\dots,m\}} \{\bar{c}_i(s_{i+1})\}, \;\; i = 1,\dots,n.$$

In any layer, the subdiagrams rooted at the nodes are isomorphic, and the decision diagram can be reduced as in Fig. 8.5(b). There is now one state in each period rather than m. If we call this state 1, the transition function is

$$g_i(1,x_i') = \begin{cases} 1, & \text{if } 0 \le x_i' \le m \\ \hat{0}, & \text{otherwise.} \end{cases}$$

The immediate cost is

$$h_i(1,x_i') = c_i(d_i + x_i' - m) + h_{i+1}x_i' + c_{i+1}(m - x_i'),$$

which can be rearranged to reveal the dependence on x_i':

$$h_i(1,x_i') = (c_i + h_{i+1} - c_{i+1})x_i' + c_i d_i + (c_{i+1} - c_i)m. \tag{8.7}$$

If $\bar{x}'_1, \ldots, \bar{x}'_n$ are the optimal controls, the resulting stock levels are given by $s_{i+1} = \bar{x}'_i$ and the production levels by $x_i = \bar{x}'_i - s_i + d_i$.

The decision diagram is therefore reduced in size by a factor of m, and solution of the problem becomes trivial. The optimal control x'_i is the one that minimizes the immediate cost (8.7), which is

$$x'_i = \begin{cases} 0 & \text{if } c_i + h_{i+1} \geq c_{i+1} \\ m & \text{otherwise.} \end{cases}$$

The optimal solution is therefore a "bang-bang" inventory policy that either empties or fills the warehouse in the next period. The optimal production schedule is

$$x_i = \begin{cases} d_i - s_i & \text{if } c_i + h_{i+1} \geq c_{i+1} \\ m + d_i - s_i & \text{otherwise.} \end{cases}$$

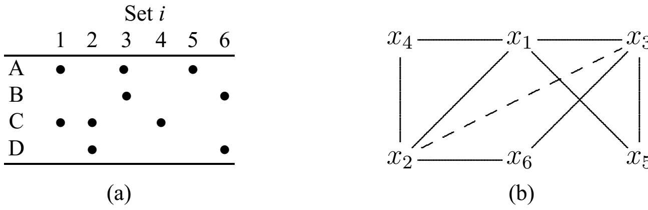
This result relies on the fact that the unit production and holding costs are linear, and excess inventory can be sold ($x_i < 0$). If excess inventory cannot be sold (or if the salvage value is unequal to the production cost), some reduction of the decision diagram is still possible, because subdiagrams rooted at states corresponding to lower inventory levels will be identical. If production and holding costs are nonlinear, the decision diagram does not simplify in general.

8.5 Nonserial Recursive Modeling

Up to this point, only *serial* recursive models have been considered. That is, the stages form a directed path in which each stage depends on the previous stage in a Markovian fashion. *Nonserial* recursions can allow one to formulate a wider variety of problems in recursive form, or to formulate a given problem using simpler states. In a nonserial recursion, the "stages" form a tree rather than a path.

Nonserial dynamic programming was introduced into operations research more than 40 years ago [29], even if it seems to have been largely forgotten in the field. Essentially the same idea has surfaced in other contexts, including Bayesian networks [109], belief logics [142, 145], pseudo-Boolean optimization [52], location theory [46], k-trees [8, 9], and bucket elimination [54].

The idea is best explained by example. Figure 8.6(a) shows a small set partitioning problem. The goal is to select a minimum subcollection of the six sets that



	Set i					
	1	2	3	4	5	6
A	•		•		•	
B			•			•
C	•	•		•		
D		•				•

(a)

Fig. 8.6 (a) A small set partitioning problem. The dots indicate which elements belong to each set i. (b) Dependency graph for the problem. The dashed edge is an induced edge.

partitions the set $\{A, B, C, D\}$, where the ith set is S_i. The control is a binary variable x_i that indicates whether S_i is selected. The feasible solutions are $(x_1, \ldots, x_6) = (0,0,0,1,1,1), (0,1,1,0,0,0), (1,0,0,0,0,1)$, where the last two solutions are optimal.

A nonserial recursion can be constructed by reference to the dependency graph for the problem, shown in Fig. 8.6(b). The graph connects two variables with an edge when the corresponding sets have an element in common. For example, x_1 and x_2 are connected because S_1 and S_2 have element C in common. Arbitrarily adopting a variable ordering $x_1, \ldots, x_6$, vertices are removed from the graph in reverse order. Each time a vertex is removed, the vertices adjacent to it are connected, if they are not already connected. Edges added in this way are *induced edges*. For example, removing x_6 in the figure induces the edge (x_2, x_3).

The feasible values of x_i depend on the set of variables to which x_i was adjacent when removed. Thus x_5 depends on (x_1, x_3), and similarly for the other control variables. Let $S_i(x_i)$ be S_i if $x_i = 1$ and the empty set otherwise. The state s^5 on which x_5 depends can be regarded as the multiset that results from taking the union $S_1(x_1) \cup S_3(x_3)$. Thus if $(x_1, x_3) = (1,1)$, the state is $\{A, A, B, C\}$. Note that A is repeated because it occurs in both S_1 and S_3. A state is feasible if and only if it contains no repeated elements.

The "stages" of a nonserial recursion correspond to the control variables as in a serial recursion. However, applying a control can result in a transition to states in multiple stages. The transition function is therefore written as $g_{ik}(s^i, x_i)$, which indicates the state in stage k that results from applying control x_i in state s^i. There is also a terminal stage, which can be denoted stage τ and allows two states, the infeasible state $\hat{0}$ and a feasible state $\hat{1}$. Only one state is possible in stage 1, namely $s^1 = \emptyset$. The transition functions for the example are

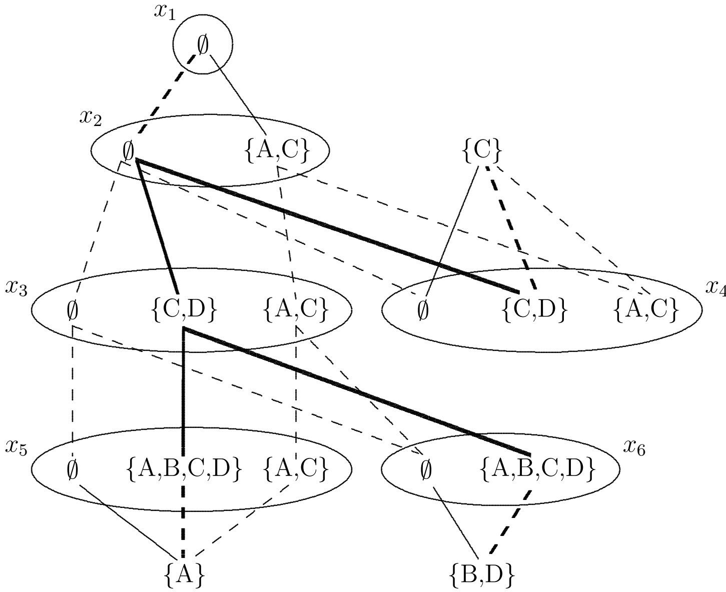


Fig. 8.7 Nonserial state transition graph for a set partitioning problem. Only nodes and arcs that are part of feasible solutions are shown. Each feasible solution corresponds to a tree incident to the root and the three terminal states, which are not encircled. The boldface tree corresponds to optimal solution $(x_1,\ldots,x_6) = (0,1,1,0,0,0)$.

$$
\begin{aligned}
&g_{12}(\emptyset,x_1) = S_1(x_1)\\
&g_{23}(s^2,x_2) = g_{24}(s^2,x_2) = s^2 \cup S_2(x_2)\\
&g_{35}(s^3,x_3) = g_{36}(s^3,x_3) = s^3 \cup S_3(x_3)\\
&g_{i\tau}(s^i,x_i) = s^i \cup S_i(x_i), \;\; i = 4,5,6
\end{aligned}
$$

with immediate cost $h_{ik}(s^i,x_i) = x_i$.

The state transition graph for the example appears in Fig. 8.7. Only feasible states are shown. States within an oval belong to the same "stage," which is labeled by the corresponding control variable x_i. The three states that are not encircled are terminal states. Arcs corresponding to a given control may run from from a state to several other stages. For example, applying control $x_3 = 1$ in state $\{\mathrm{C},\mathrm{D}\}$ creates arcs to state $\{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D}\}$ in both stages 5 and 6.

Feasible solutions correspond to trees that are incident to the initial state and the three terminal states. This contrasts with serial decision diagrams, in which solutions are paths incident to the initial state and terminal state. The tree shown in bold is one of the two optimal solutions. Note that its cost is 2 even though it contains four solid arcs, because solid arcs leaving a state s^i correspond to the same choice $x_i = 1$.

The state transition graph of Fig. 8.7 can be regarded as a *nonserial decision diagram*. The ovals correspond to “layers,” and the three terminal states belong to the terminal layer. In general, the concepts of reduction and canonical costs can be carried over from serial to nonserial decision diagrams.

Chapter 9
MDD-Based Constraint Programming

Abstract This is the first of three chapters that apply decision diagrams in the context of constraint programming. This chapter starts by providing a background of the solving process of constraint programming, focusing on consistency notions and constraint propagation. We then extend this methodology to MDD-consistency and MDD-based constraint propagation. We present MDD propagation algorithms for specific constraint types, including linear inequalities, ALLDIFFERENT, AMONG, and ELEMENT constraints, and experimentally demonstrate how MDD propagation can improve conventional domain propagation.

9.1 Introduction

The previous chapters have focused on developing decision diagrams for optimization. For example, in Chapter 6 we saw how decision diagrams can form the basis for a stand-alone solver for discrete optimization problems. In the following chapters we take a different perspective, and integrate decision diagrams within a constraint programming framework. In particular, we will discuss how multivalued decision diagrams (MDDs) can be used to improve constraint propagation, which is the central inference process of constraint programming.

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_9

9.2 Constraint Programming Preliminaries

Constraint programming (CP) provides a modeling and solving environment for continuous and discrete optimization problems [137]. Perhaps the most distinctive feature of CP, compared with similar frameworks such as integer linear programming (ILP) and Boolean satisfiability (SAT), is its versatile modeling language. For example, variables are not restricted to take a value from a numeric interval (as in ILP) or from a Boolean domain (as in SAT), but can be assigned an element from any finite set. Moreover, whereas problems must be represented via linear constraints in ILP or Boolean clauses in SAT, CP allows constraints to be any relation over a finite set of variables. These constraints can be algebraic or logical expressions, be expressed as an explicit list of allowed tuples, and may even take a symbolic form such as $\text{ALLDIFFERENT}(x_1, x_2, \ldots, x_n)$ which specifies that variables $x_1, x_2, \ldots, x_n$ take distinct values. Such symbolic constraints are referred to as *global constraints* in the CP literature [151, 133].

As a consequence of the rich modeling language of CP systems, the solving process cannot exploit the uniform semantics of the representation, like linear programming relaxations in ILP. Instead, CP solvers combine a systematic search process with dedicated inference techniques for each constraint type. That is, the solver associates with each constraint in the model a so-called *propagation algorithm*, whose role is to remove provably inconsistent values from the domain of possible values for each variable in the scope of the constraint [32]. This is also referred to as *domain filtering* in the literature. The *constraint propagation* process then considers each of the constraints in turn until no more domain filtering takes place. A CP solver typically applies constraint propagation at each search state, and as a result of the domain filtering the search space can be drastically reduced. We first illustrate this process in Example 9.1, and then provide a formal description.

Example 9.1. Consider the following CP model:

$$
\begin{aligned}
& x_1 > x_2 && (c_1) \\
& x_1 + x_2 = x_3 && (c_2) \\
& \text{ALLDIFFERENT}(x_1, x_2, x_3, x_4) && (c_3) \\
& x_1 \in \{1,2\}, x_2 \in \{0,1,2,3\}, x_3 \in \{2,3\}, x_4 \in \{0,1\}
\end{aligned}
$$

This model is a constraint satisfaction problem (CSP) as it does not have an objective function to be optimized. We apply constraint propagation by considering each

constraint in turn. From constraint (c_1) we deduce that $x_2 \in \{0,1\}$. Then we consider constraint (c_2), but find that each domain value for each variable participates in a solution to the constraint. When we consider constraint (c_3), we can realize that the values $\{0,1\}$ will be assigned (in any order) to variables x_2 and x_4, and hence we reduce the domain of x_1 to $\{2\}$, and consequently $x_3 \in \{3\}$. We continue revisiting the constraints whose variables have updated domains. Constraint (c_1) does not reduce any more domains, but by constraint (c_2) we deduce $x_2 \in \{1\}$. Constraint (c_3) then updates $x_4 \in \{0\}$. No additional domain filtering is possible, and we finish the constraint propagation process. In this case, we arrive at a solution to the problem, $(x_1,x_2,x_3,x_4) = (2,1,3,0)$.

Domain Consistency

It is useful to characterize the outcome of the constraint propagation process. Let $C(x_1,x_2,\ldots,x_n)$ be a constraint, and let each variable x_i have an associated domain of possible values $D(x_i)$, for $i = 1,\ldots,n$. A solution to C is a tuple $(v_1,v_2,\ldots,v_n)$ such that $v_i \in D(x_i)$ and that satisfies C. If for all x_i and all $v_i \in D(x_i)$ there exists a solution to C in which $x_i = v_i$ $(i = 1,\ldots,n)$, we say that C is *domain consistent*. We say that a set of constraints $\{C_1(X_1),C_2(X_2),\ldots,C_m(X_m)\}$ with respective scopes $X_1,X_2,\ldots,X_m$ is domain consistent if all individual constraints are.[1] Observe that domain consistency is defined relative to the domains of the variables in the scope of the constraint. In the remainder, we will assume that a variable x implicitly defines its associated domain $D(x)$, unless otherwise noted.

We can establish domain consistency for a constraint $C(X)$ by determining for each variable–value pair whether it belongs to a solution to $C(X)$, as follows:

1: DOMAINCONSISTENCY($C(X)$)
2: **for** $x \in X$ **do**
3:     **for** $v \in D(x)$ **do**
4:         **if** C has no solution with $x = v$ **then**
5:             $D(x) := D(x) \setminus \{v\}$
6:             **if** $D(x) = \varnothing$ **then**
7:                 **return false**
8: **return true**

[1] Note that, even if a set of constraints is domain consistent, the *conjunction* of the constraints in the set may not be domain consistent.

This function will return ‘true’ if the constraint is domain consistent, and returns ‘false’ otherwise. In the latter case one of the domains is empty, hence no solution exists, and the CP solver can backtrack from the current search state. The time complexity of algorithm DOMAINCONSISTENCY is polynomial in the number of variables and the size of the variable domains, but relies on the time complexity for determining whether a solution exists to the constraint. For some constraints this can be done in polynomial or even constant time, while for others the check for feasibility is an NP-complete problem itself (see Example 9.4).

Example 9.2. A classical way of representing CSPs is by means of so-called *table* constraints, which provide an explicit list of allowed tuples for a set of variables. A special case is a binary table constraint, which is defined on two variables (whose domains can be any finite set). We can adapt algorithm DOMAINCONSISTENCY to establish domain consistency on a binary constraint $C(x_1,x_2)$ as follows:

1: DOMAINCONSISTENCYBINARYTABLE($C(x_1,x_2)$)
2: Let $b_1(v) :=$ **false** for all $v \in D(x_1)$
3: Let $b_2(v) :=$ **false** for all $v \in D(x_2)$
4: **for** $(v_1,v_2) \in C(x_1,x_2)$ **do**
5: $b_1(v_1) :=$ **true**, $b_2(v_2) :=$ **true**
6: $D(x_1) := D(x_1) \setminus \{v \mid v \in D(x_1), b_1(v) = \textbf{false}\}$
7: **if** $D(x_1) = \varnothing$ **then**
8: **return false**
9: $D(x_2) := D(x_2) \setminus \{v \mid v \in D(x_2), b_2(v) = \textbf{false}\}$
10: **if** $D(x_2) = \varnothing$ **then**
11: **return false**
12: **return true**

Here, the Boolean parameters $b_1(v)$ and $b_2(v)$ represent whether a domain value v in $D(x_1)$, resp. $D(x_2)$, participates in a solution to C or not. Algorithm DOMAINCONSISTENCYBINARYTABLE considers all tuples in C and therefore runs in time and space $O(|C(x_1,x_2)|)$.

We note that much more refined variants of this algorithm exist in the literature; see for example [132].

The generic algorithm DOMAINCONSISTENCY can be made more efficient by considering the semantics of the constraint it is applied to; in some cases it is not necessary to consider all variable–value pairs, as illustrated in the following example:

Example 9.3. Consider a linear inequality constraint of the form

$$\sum_{i=1}^{n} w_i x_i \leq U$$

for numeric variables x_i, nonnegative constants w_i ($i = 1, \ldots, n$), and a constant right-hand side U. This constraint can be made domain consistent in polynomial time by updating each variable domain $D(x_i)$, $i = 1, \ldots, n$, according to the following rule:

$$D(x_i) := D(x_i) \setminus \left[(U - \sum_{j \neq i} w_j \min D(x_j))/w_i, \max D(x_i) \right].$$

In practice, this rule can be implemented by updating the maximum value in $D(x_i)$, which may be done in constant time with the appropriate data structure for storing the domains.

The following example illustrates that establishing domain consistency can be NP-hard for certain constraint types:

Example 9.4. The UNARYRESOURCE constraint was introduced in CP systems to represent the problem of scheduling a set A of activities on a single, unit-capacity machine (non-preemptively). Each activity $a \in A$ has a given release time r_a, deadline d_a, and processing time p_a. We introduce a set of variables $S = \{s_a \mid a \in A\}$, where s_a represents the start time of activity a. Then the constraint

$$\text{UNARYRESOURCE}(S, \mathbf{r}, \mathbf{d}, \mathbf{p})$$

is feasible if and only if the start times in S correspond to a feasible schedule. Since this constraint represents an NP-complete problem [71], there is no known polynomial-time algorithm to determine whether there exists a solution, let alone establish domain consistency.

Bounds Consistency

Domain consistency is one of the stronger notions of consistency one can apply to an individual constraint. However, as mentioned above, for some constraints establishing domain consistency may be NP-hard. For others, domain consistency may be too time-consuming to establish, relative to the associated domain reduction.

Therefore, other consistency notions have been introduced that are weaker, but are more efficient to establish. The most important alternative is that of *bounds consistency*.

Let $C(x_1, x_2, \ldots, x_n)$ be a constraint, and let each variable x_i have an associated domain $D(x_i)$ that is a subset of a totally ordered universe U. We say that C is *bounds consistent* if for all x_i and $v_i \in \{\min D(x_i), \max D(x_i)\}$, there exists a solution $(v_1, v_2, \ldots, v_n)$ such that $v_j \in [\min D(x_j), \max D(x_j)]$, $j \neq i$. In other words, we consider a convex relaxation of the variable domains, and require that all bounds participate in a solution to C.

Similar to domain consistency, bounds consistency can be established via a generic algorithm, by adapting algorithm DOMAINCONSISTENCY to consider domain bounds. But oftentimes, more efficient algorithms exist for specific constraint types.

Example 9.5. Consider again the linear inequality constraint from Example 9.3. For this constraint, establishing bounds consistency is equivalent to establishing domain consistency.

Constraint Propagation Cycle

We mentioned before that a set of constraints is domain consistent if all individual constraints are. We can establish domain consistency on a set of constraints $\{C_1(X_1), C_2(X_2), \ldots, C_m(X_m)\}$ using a constraint propagation process, as follows:

1: CONSTRAINTPROPAGATION($\{C_1(X_1), C_2(X_2), \ldots, C_m(X_m)\}$)
2: $Q := \{C_1(X_1), C_2(X_2), \ldots, C_m(X_m)\}$
3: **while** Q is not empty **do**
4:     Let $C(X)$ be an element from Q, and delete it from Q
5:     **if** DOMAINCONSISTENCY($C(X)$) = **false then**
6:         **return false**
7:     **for** $x \in X$ such that $D(x)$ has been updated **do**
8:         $Q := Q \cup \{C_j(X_j) \mid j = 1, \ldots, m, C_j \neq C, x \in X_j\}$
9: **return true**

Algorithm CONSTRAINTPROPAGATION maintains a set (or queue) Q, which is initialized by the set of constraints $\{C_1, \ldots, C_m\}$. It iteratively considers an element from Q (line 4), and makes it domain consistent (line 5). If any variable domain

$D(x)$ is updated by this process, each constraint that has variable x in its scope is added again to Q, to be revisited (lines 7–8). If algorithm DOMAINCONSISTENCY detects an empty domain, it returns 'false', as will CONSTRAINTPROPAGATION (lines 5–6). Otherwise, CONSTRAINTPROPAGATION returns 'true' (line 9). Practical implementations of CONSTRAINTPROPAGATION can be made more efficient, for example by choosing the order in which constraints are processed. We already saw this algorithm in action in Example 9.1, where we repeatedly established domain consistency on the individual constraints until no more domain reductions were possible.

The constraint propagation process can be adapted to include bounds consistency algorithms instead of domain consistency algorithms for specific constraints. Regardless, when the variables have finite domains, the propagation cycle is guaranteed to terminate and reach a fixed point. Moreover, under certain conditions of the propagators, one can show that the fixed point is unique, irrespective of the order in which the propagators are applied [7].

Systematic Search

The systematic search process is an important element of CP solvers, and many CP systems allow user-specified search strategies. Some systems, such as Objective-CP [149], have a modeling language with dedicated syntax to 'model' the search process, while other systems, such as IBM ILOG CPO, have a more restricted user interaction and rely on automated search heuristics [155].

Modern finite-domain CP solvers implement a depth-first search strategy. A classical search strategy is to enumerate variable–value assignments, and apply constraint propagation at each search state to detect infeasible subproblems that can be discarded. Such enumeration procedures require a variable selection heuristic, and a value selection heuristic. The associated heuristic scoring functions are often dynamically computed during the search. Search decisions can also be made based on constraints; for example, we can partition the search space via $x \leq y$ or $x > y$ for two variables x, y. Regardless of the search strategy, CP solvers apply constraint propagation at each search state, usually until a fixed point is reached. A detailed description of classical CP search strategies can be found in [148]. An overview of results with respect to learning during search can be found in [131].

In the context of optimization problems, the default behavior of CP systems is to regard the objective as a constraint. For example, if the goal is to minimize an

objective z, we impose the constraint $z \leq U$ for an upper bound U, which is updated each time we encounter a better solution during search. The CP literature contains many examples of much stronger optimization approaches, however [150].

9.3 MDD-Based Constraint Programming

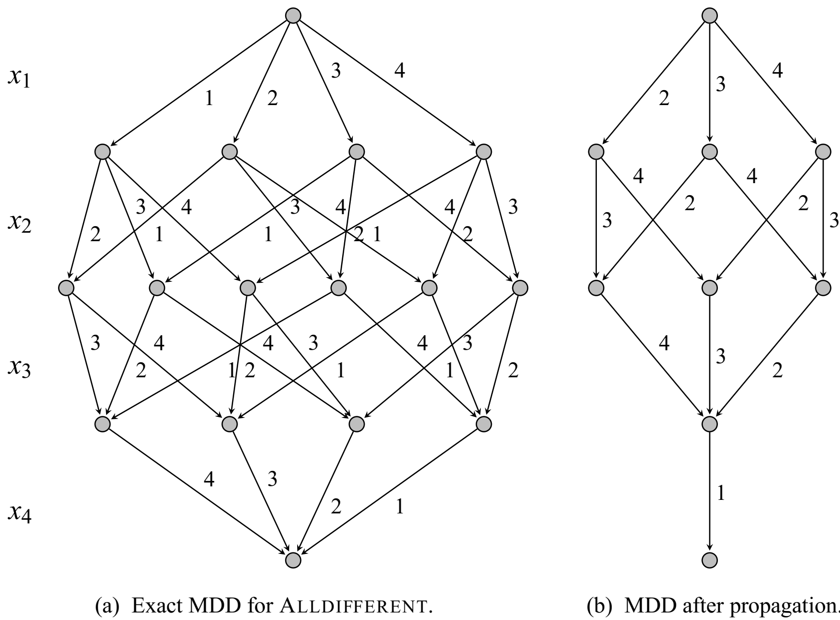
During the constraint propagation process information is communicated between constraints through the variable domains. That is, all inference that takes place within a constraint must first be projected onto the variable domains before it can be passed to other constraints. This form of constraint propagation is said to rely on the *domain store* representation. A weakness of the domain store is that it accounts for no interaction between variables, and any solution in the Cartesian product of the variable domains is consistent with it. The following example illustrates this weakness, and also suggests that stronger inference may be worthwhile to pursue.

Example 9.6. Consider the following CSP:

$$\begin{array}{ll} \text{ALLDIFFERENT}(x_1,x_2,x_3,x_4) & (c_1) \\ x_1 + x_2 + x_3 \geq 9 & (c_2) \\ x_i \in \{1,2,3,4\}, i = 1,\ldots,4 & \end{array}$$

Both constraints are domain consistent. However, if we were able to communicate from (c_1) to (c_2) that variables x_1,x_2,x_3 must take distinct values, we would be able to deduce that these three variables cannot take value 1, i.e., $x_1,x_2,x_3 \in \{2,3,4\}$, and similarly $x_4 \in \{1\}$.

A brute-force approach to deducing the inference in Example 9.6 is to introduce a new constraint type to the CP system that represents the conjunction of linear inequalities and ALLDIFFERENT constraints. This idea can be extended to conjunctions of arbitrary sets of constraints, which will however quickly lead to scalability issues. After all, the philosophy behind constraint programming is that constraints (or combinatorial substructures) are processed individually. We therefore follow a different approach, which is to improve the communication between constraints by passing more structural information than the domain store, in the form of limited-width MDDs [4]. In other words, we replace (or augment) the domain store with an *MDD store*.



(a) Exact MDD for ALLDIFFERENT.

(b) MDD after propagation.

Fig. 9.1 MDD propagation for Example 9.7.

Example 9.7. We continue Example 9.6. In the ideal case, we would create an exact MDD to represent constraint (c_1), as in Fig. 9.1(a). We then communicate the MDD, which represents relationships between the variables, to constraint (c_2). We can inspect that all paths in the MDD that use an arc with label 1 for x_1, x_2, x_3 do not satisfy constraint (c_2), and hence these arcs can be removed. We can also remove nodes that are no longer connected to either the root or the terminal. The resulting MDD is depicted in Fig. 9.1(b). If we project this information to the variable domains, we obtain $x_1, x_2, x_3 \in \{2,3,4\}$ and $x_4 \in \{1\}$, which can be communicated to the domain store.

Observe that in Example 9.7 the exact MDD for the ALLDIFFERENT constraint has exponential size. In practice one therefore needs to use *relaxed* MDDs, of limited width, to make this approach scalable.

In the remainder of this chapter, we will assume that the MDD defined on a set of variables $\{x_1, x_2, \ldots, x_n\}$ will follow the lexicographic ordering of the variables, unless noted otherwise. Also, we assume that the MDDs do not contain long arcs in this chapter.

9.3.1 MDD Propagation

In principle, the MDD store can replace the domain store completely, and we could have one single MDD representing the entire problem. This may not be efficient in practice, however, as the resulting relaxed MDD may not be sufficiently strong. Instead we recommend to maintain both the domain store and an MDD store, where the MDD store represents a suitable substructure and not necessarily all aspects of the problem. Moreover, we may choose to introduce multiple MDDs, each representing a substructure or group of constraints that will be processed on it. Consider for example a workforce scheduling problem in which we need to meet given workforce levels over time, while meeting individual employee workshift rules. We may choose to introduce one MDD per employee, representing the employee's work schedule over time, while the workforce-level constraints are enforced using a conventional CP model with domain propagation.

The purpose of the MDD store is to represent a more refined relationship among a set of variables than the domain store's Cartesian product of variable domains. This is accomplished by MDD *filtering* and *refinement*. MDD filtering generalizes the concept of domain filtering to removing infeasible arcs from an MDD, while the refinement attempts to strengthen the MDD representation by splitting nodes, within the allowed maximum width. We note that a principled approach to node refinement in MDDs was introduced by Hadzic et al. [84]. A generic top-down filtering and refinement compilation scheme for a given set of constraints was presented in Section 4.7, as Algorithm 4. This is precisely the procedure we will apply in the context of MDD-based constraint programming. Note that we can adjust the strength of the MDD store by setting the maximum allowed MDD width from one (the domain store) to infinity (exact MDD).

Example 9.8. Consider a CSP with variables $x_1 \in \{0,1\}, x_2 \in \{0,1,2\}, x_3 \in \{1,2\}$, and constraints $x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3$. All domain values are domain consistent (even if we were to apply the ALLDIFFERENT propagator on the conjunction of the constraints), and the coventional domain store defines the relaxation $\{0,1\} \times \{0,1,2\} \times \{1,2\}$, which includes infeasible solutions such as $(1,1,1)$.

The MDD-based approach starts with the MDD of width one in Fig. 9.2(a), in which parallel arcs are represented by a set of corresponding domain values for clarity. We refine and propagate each constraint separately. Starting with $x_1 \neq x_2$, we refine the MDD by splitting the node at layer 2, resulting in Fig. 9.2(b). This allows us to delete two domain values, based on $x_1 \neq x_2$, as indicated in the figure. In

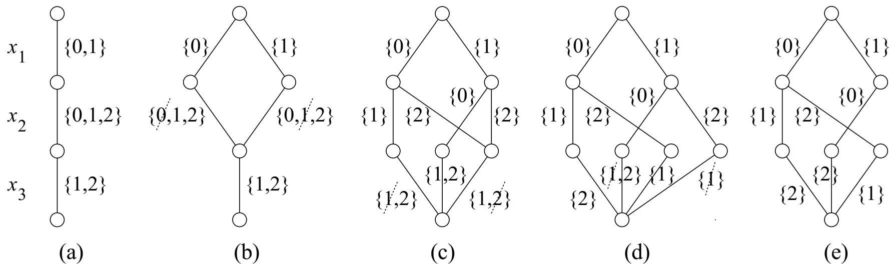


Fig. 9.2 Refining and propagating an MDD of width one (a) for $x_1 \neq x_2$ (b), $x_2 \neq x_3$ (c), and $x_1 \neq x_3$ (d), yielding the MDD in (e). Dashed lines mark removed inconsistent values.

Fig. 9.2(c) and (d) we refine and propagate the MDD for the constraints $x_2 \neq x_3$ and $x_1 \neq x_3$, respectively, until we reach the MDD in Fig. 9.2(e). This MDD represents all three solutions to the problem, and provides a much tighter relaxation than the Cartesian product of variable domains.

We can integrate MDD propagation within a CP system by adjusting the propagation algorithms for constraints of our choice to operate on MDDs. That is, instead of variable domains, these constraints now receive an arbitrary MDD, to be filtered and refined. For each constraint type the MDD propagation algorithm will be a tailored version of Algorithm 4, which can directly be inserted in the CONSTRAINTPROPAGATION procedure.

9.3.2 MDD Consistency

We will next discuss how the outcome of MDD propagation can be characterized. Let $C(X)$ be a constraint with variables X as its scope, and let MDD M be defined on a set of variables $X' \supseteq X$. A path with arc labels $v_1,\ldots,v_n$ from the root $\mathbf{r}$ to the terminal $\mathbf{t}$ in M is feasible for C if setting $x_i = v_i$ for all $x_i \in X$ satisfies C. Following [4], we say that C is *MDD consistent* with respect to M if every arc in M belongs to a path that is feasible for C.

Note that domain consistency is equivalent to MDD consistency on an MDD of width one, in which the nodes of subsequent layers L_i and L_{i+1} are connected with parallel arcs with labels $D(x_i)$.

As seen in Section 4.7, the feasibility of an MDD arc $a = (u,v)$ in layer L_j with respect to constraint C can be determined by the transition function $t_j^C(s(u),d(a))$.

Recall that $s(u)$ represents the (top-down) state information of node u and $d(a)$ represents the label of a. Note that parallel arcs are distinguished uniquely by their labels. In addition, MDD propagation algorithms may utilize bottom-up state information, as we have seen in Section 4.7.1. For clarity, we will describe the MDD propagation rules for an arc $a = (u,v)$ in terms of $s^{\downarrow}(u)$, $s^{\uparrow}(v)$, and $d(a)$, where $s^{\downarrow}(\cdot)$ and $s^{\uparrow}(\cdot)$ represent the state information computed from the root **r** and terminal **t**, respectively. For each constraint type, the MDD propagator is based on an appropriate definition of state information, and a recursive procedure to compute this information during the top-down and bottom-up pass.

For some constraints we can establish MDD consistency, with respect to a given MDD, in polynomial time. We will list some of these constraints in later sections. In general, if we can determine in polynomial time whether any particular variable–value assignment is consistent with the MDD and a constraint C, then we can achieve MDD consistency in polynomial time due to the following theorem:

Theorem 9.1. *Suppose that the feasibility of $x_j = v$ for a given constraint C on a given MDD M can be determined in $O(f(M))$ time and space for any variable x_j in the scope C and any $v \in D(x_j)$. Then we can achieve MDD consistency for C in time and space at most $O(\text{poly}(M)f(M))$.*

Proof. The proof is based on a shaving argument. For each arc a of M we consider the MDD M_a that consists of all the **r**–to–**t** paths in M containing a. Then a can be removed from M if and only if $x_j = d(a)$ is inconsistent with C and M_a, where j is the layer from which a originates. This can be determined in time and space at most $O(f(M_a)) \leq O(f(M))$. By repeating this operation for each arc of M we obtain the theorem. □

To establish MDD consistency in practice, the goal is to efficiently compute state information that is strong enough to apply Theorem 9.1. In the sequel, we will see that this can be done for inequality constraints and AMONG constraints in polynomial time, and in pseudo-polynomial time for two-sided inequality constraints. Furthermore, we have the following result:

Corollary 9.1. *For binary table constraints, MDD consistency can be established in polynomial time (in the size of the constraint and the MDD).*

Proof. Let $C(x_i,x_j)$ be a binary table constraint, where $i < j$ without loss of generality. We let state $s^{\downarrow}(u)$ contain all the values assigned to x_i in some path from

$\mathbf{r}$ to u, and $s^{\uparrow}(u)$ contain all the values assigned to x_j in some path from u to $\mathbf{t}$. We initialize $s^{\downarrow}(\mathbf{r}) = \emptyset$, and recursively compute

$$s^{\downarrow}(v) = \begin{cases} \cup_{a'=(u,v)\in in(v)} \; d(a') \text{ if } u \in L_i \\ \cup_{a'=(u,v)\in in(v)} \; s^{\downarrow}(u) \text{ otherwise} \end{cases}$$

for all nodes v in layers $L_2, \ldots, L_{n+1}$. Similarly, we initialize $s^{\downarrow}(\mathbf{t}) = \emptyset$, and recursively compute

$$s^{\uparrow}(u) = \begin{cases} \cup_{a'\in out(u)} \; d(a') \qquad \text{if } u \in L_j \\ \cup_{a'=(u,v)\in out(u)} \; s^{\uparrow}(v) \text{ otherwise} \end{cases}$$

for all nodes u in layers $L_n, L_{n-1}, \ldots, L_1$. Then we can delete arc $a = (u,v)$ from M if and only if

(a) $u \in L_i$ and tuple $(d(a), w)$ does not satisfy C for any $w \in s^{\uparrow}(v)$, or
(b) $v \in L_j$ and tuple $(w, d(a))$ does not satisfy C for any $w \in s^{\downarrow}(u)$.

The top-down and bottom-up passes to compute the states and perform the checks for feasibility require time that is polynomial in the size of M. The corollary then follows from Theorem 9.1. □

9.3.3 MDD Propagation by Intersection

In the CP literature exact MDDs have been used to represent the solution set for specific constraints. In fact, there is a close relationship between MDDs, table constraints, and the so-called REGULAR constraint that represents fixed-length strings that belong to a regular language [126, 20]. Namely, they provide different data structures for storing explicit lists of tuples that satisfy the constraint; see [43] for a computational comparison, and [123, 124] for efficient domain consistency algorithms based on MDDs. In such cases, we want to establish MDD consistency of one MDD with respect to another. We will next see that this can be done in a generic fashion. Moreover, it will provide us with another tool to test whether MDD consistency can be established in polynomial time for certain constraint types.

Formally, our goal in this section is to establish MDD consistency on a given MDD M with respect to another MDD M' on the same set of variables. That is, M is MDD consistent with respect to M' if every arc in M belongs to a path (solution)

Algorithm 7 Intersection(M,M')

Input: MDD M with root r, MDD M' with root r'. M and M' are defined on the same ordered sequence of n variables.
Output: MDD I with layers $L_1^I, \ldots, L_{n+1}^I$ and arc set A^I. Each node u in I has an associated state $s(u)$.

1: create node r^I with state $s(r^I) := (r, r')$
2: $L_1^I := \{r^I\}$
3: **for** $i = 1$ **to** n **do**
4: $\quad L_{i+1}^I := \{\}$
5: $\quad$ **for all** $u \in L_i^I$ with $s(u) = (v, v')$ **do**
6: $\quad\quad$ **for all** $a = (v, \tilde{v}) \in M$ and $a' = (v', \tilde{v}') \in M'$ such that $d(a) = d(a')$ **do**
7: $\quad\quad\quad$ create node $\tilde{u}$ with state $s(\tilde{u}) := (\tilde{v}, \tilde{v}')$
8: $\quad\quad\quad$ **if** $\exists\, \tilde{w} \in L_{j+1}^I$ with $s(\tilde{w}) = s(\tilde{u})$ **then** $\tilde{u} := \tilde{w}$
9: $\quad\quad\quad$ **else** $L_{i+1}^I \mathrel{+}= \tilde{u}$ **end if**
10: $\quad\quad\quad$ add arc $(u, \tilde{u})$ with label $d(a)$ to arc set A^I
11: remove all arcs and nodes from I that are not on a path from r^I to $t^I \in L_{n+1}^I$
12: **return** I

Algorithm 8 MDD-Consistency(M,M')

Input: MDD M with root r, MDD M' with root r'. M and M' are defined on the same ordered sequence of n variables.
Output: M that is MDD consistent with respect to M'

1: create I := Intersection(M,M')
2: **for** $i = 1$ **to** n **do**
3: $\quad$ create array Support$[u, l] := 0$ for all $u \in L_i^M$ and arcs out of u with label l
4: $\quad$ **for all** arcs $a = (v, \tilde{v})$ in A^I with $s(v) = (u, u')$ such that $v \in L_i^I$ **do**
5: $\quad\quad$ Support$[u, d(a)] := 1$
6: $\quad$ **for all** arcs $a = (u, \tilde{u})$ in M such that $u \in L_i^M$ **do**
7: $\quad\quad$ **if** Support$[u, d(a)] = 0$ **then** remove a from M **end if**
8: remove all arcs and nodes from M that are not on a path from r to $t \in L_{n+1}^M$
9: **return** M

of M that also exists in M'. For our purposes, we assume that M and M' follow the same variable ordering.

We can achieve MDD consistency by first taking the intersection of M and M', and then removing all arcs from M that are not compatible with the intersection. Computing the intersection of two MDDs is well studied, and we present a top-down intersection algorithm that follows our definitions in Algorithm 7. This description is adapted from the ‘melding’ procedure in [104].

The intersection MDD, denoted by I, represents all possible paths (solutions) that are present in both M and M'. Each *partial* path in I from the root r^I to a node

u thus will exist in M and M', with respective endpoints v, v'. This information is captured by associating with each node u in I a state $s(u) = (v, v')$ representing those nodes $v \in M$ and $v' \in M'$. The root of I is initialized as r^I with $s(r^I) := (r, r')$, where r and r' are the respective roots of M and M' (lines 1–2). The algorithm then, in a top-down traversal, considers a layer L_i^I in I, and augments a node $u \in L_i^I$ with $s(u) = (v, v')$ with an arc only if both M and M' have an arc with the same label out of v and v', respectively (lines 5–7). If the next layer already contains a node $\tilde{u}$ with the same state, we reuse that node. Otherwise we add a new node $\tilde{u}$ to L_{i+1}^I and add the arc $(u, \tilde{u})$ to I. Note that the last layer of I contains a single terminal t^I with state $s(t^I) = (t, t')$, provided that I is not empty. In the last step (line 14) we clean up I by removing all arcs and nodes that do not belong to a feasible path. This can be done in a bottom-up traversal of I. Observe that this algorithm does not necessarily create a *reduced* MDD.

Algorithm 8 presents an algorithm to establish MDD consistency on M with respect to M'. We first compute the intersection I of M and M' (line 1). We then traverse M in a top-down fashion, and for each layer L_i^M we identify and remove infeasible arcs. For this, we define a Boolean array Support$[u, l]$ (initialized to 0) that represents whether an arc out of node $u \in M$ with label l has support in I (line 3). In line 4, we consider all arcs out of layer L_i^I in I. If an arc $a = (v, \tilde{v})$ exists in L_i^I with label l and $s(v) = (u, u')$, we mark the associated arc out of u as supported by setting Support$[u, l] := 1$ (lines 4–6). We then remove all arcs out of L_i^M that have no support (lines 7–9). Lastly, we again clean up M by removing all arcs and nodes that do not belong to a feasible path (line 11).

Theorem 9.2. *Algorithm 8 establishes MDD consistency on M with respect to M' in $O(|M| \cdot \omega(M'))$ time and space, where $|M|$ is the number of nodes in M and $\omega(M')$ is the width of M'.*

Proof. The correctness of Algorithm 7 follows by induction on the number of layers. To prove that Algorithm 8 establishes MDD consistency, consider an arc $a = (u, \tilde{u})$ in M after applying the algorithm. There exists a node $v \in I$ with $s(v) = (u, u')$ such that solutions represented by the paths from r to u in M and from r' to u' in M' are equivalent. There also exists an arc $a^I = (v, \tilde{v}) \in A^I$ with the same label as a. Consider $s(\tilde{v}) = (w, w')$. Since M and I are decision diagrams, a label appears at most once on an arc out of a node. Therefore, $w = \tilde{u}$. Since a^I belongs to I, there exist paths from w (or $\tilde{u}$) to t in M and from w' to t' in M' that are equivalent. Hence, a belongs to a feasible path in M (from r to u, then along a into

$\tilde{u}$ and terminating in t) for which an equivalent path exists in M' (from r' to u', then into w' and terminating in t').

Regarding the time complexity for computing the intersection, a coarse upper bound multiplies n (line 3), $\omega(M) \cdot \omega(M')$ (line 5), and $d_{\max}^2$ (line 6), where $d_{\max}$ represents the maximum degree out of a node, or $\max_{x \in X} |D(x)|$. We can amortize these steps since the for-loops in lines 3 and 6 consider each arc in M once for comparison with arcs in M'. Each arc is compared with at most $\omega(M')$ arcs (line 6); here we assume that we can check in constant time whether a node has an outgoing arc with a given label (using an arc-label list). This gives a total time complexity of $O(|M| \cdot \omega(M'))$. The memory requirements are bounded by the size of the intersection, which is at most $O(n \cdot \omega(M) \cdot \omega(M') \cdot d_{\max}) = O(|M| \cdot \omega(M'))$. This dominates the complexity of Algorithm 8, since lines 2–12 can be performed in linear time and space (in the size of M). □

Observe that Algorithm 8 no longer ensures that each *solution* in M is represented by some path in M', as is the case for the intersection. MDD consistency merely establishes that each *arc* in M belongs to some solution that is also in M'. Although MDD intersections are stronger than MDD consistency, their limitation is that the width of the intersection MDD may be as large as the product of the widths of M and M'. Therefore intersecting M with multiple MDDs will, in general, increase the size of the resulting MDD exponentially.

Theorem 9.2 can also be applied to obtain the tractability of establishing MDD consistency on certain constraints. Namely, if we can represent a constraint $C(X)$ using an exact MDD M' of polynomial size, by Theorem 9.2 we can establish MDD consistency with respect to a given MDD M in polynomial time (provided that M and M' follow the same variable ordering).

The converse of Theorem 9.2 does not hold: there exist constraints for which MDD consistency can be achieved in polynomial time on any given MDD, while a minimal reduced exact MDD has exponential size. As a specific example, consider linear inequality constraints of the form $\sum_{i=1}^{n} a_i x_i \geq b$, where x_i is an integer variable, a_i is a constant, for $i = 1, \ldots, n$, and b is a constant. MDD consistency can be achieved for such constraints in linear time, for any given MDD, by computing for each arc the longest **r**-**t** path (relative to the coefficients a_i) that uses that arc ([4]; see also Section 9.4.2 below). However, the following linear inequality corresponds to a reduced exact MDD that grows exponentially: For k even and $n = k^2$, consider $\sum_{1 \leq i,j \leq k} a_{ij} x_{ij} \geq k(2^{2k} - 1)/2$, where x_{ij} is a binary variable, and $a_{ij} = 2^{i-1} + 2^{k+j-1}$, for $1 \leq i, j \leq k$. It is shown in [98] that, for any variable order,

the size of the reduced ordered BDD for this inequality is bounded from below by $\Omega(2^{\sqrt{n}/2})$.

9.4 Specialized Propagators

We next present MDD propagation algorithms for several constraint types. In some cases, the propagation may not be as strong as for the conventional domain store, in the sense that, when specialized to an MDD of width one, it may not remove as many values as a conventional propagator would. However, a 'weak' propagation algorithm can be very effective when applied to the richer information content of the MDD store.

If one prefers not to design a propagator specifically for MDDs, there is also the option of using a conventional domain store propagator by adapting it to MDDs. This can be done in a generic fashion, as will be explained in Section 9.4.7.

Lastly, our description will mainly focus on MDD *filtering*. The refinement process can be based on the same state information as is used for filtering.

9.4.1 Equality and Not-Equal Constraints

We first illustrate MDD propagation of the constraints $x_i = x_j$ and $x_i \neq x_j$. Because these are binary constraints (i.e., defined on two variables), we could represent them as table constraints and apply Corollary 9.1 to establish MDD consistency. By taking into account the semantics of the constraints, however, there is no need to use explicit tables. Namely, using the states $s^{\downarrow}(u)$ and $s^{\uparrow}(u)$ as defined in the proof of Corollary 9.1, we can achieve MDD consistency for $x_i = x_j$ by deleting an arc $a = (u,v)$ whenever $u \in L_i$ and $d(a) \notin s^{\uparrow}(v)$, and $v \in L_j$ and $d(a) \notin s^{\downarrow}(u)$. Likewise, we can achieve MDD consistency for $x_i \neq x_j$ by deleting an arc $a = (u,v)$ whenever $u \in L_i$ and $s^{\uparrow}(v) = \{d(a)\}$, and $v \in L_j$ and $s^{\downarrow}(u) = \{d(a)\}$.

Observe that this procedure generalizes directly to propagating $f_i(x_i) = f_j(x_j)$ and $f_i(x_i) \neq f_j(x_j)$ for functions f_i and f_j. The scheme can also be applied to constraints $x_i < x_j$. However, in this case we only need to maintain bound information instead of sets of domain values, which leads to an even more efficient implementation.

9.4.2 Linear Inequalities

We next focus on the propagation algorithm for general inequalities, as proposed in [4]. That is, we want to propagate an inequality over a separable function of the form

$$\sum_{j \in J} f_j(x_j) \leq b. \tag{9.1}$$

We can propagate such a constraint on an MDD by performing shortest-path computations.

The length of an arc $a = (u,v)$ with $u \in L_j$ is defined as $l_a := f_j(d(a))$ if $j \in J$, and $l_a := 0$ otherwise. Thus the length of an **r**–to–**t** path is the left-hand side of (9.1).

For a node u in the MDD, let $sp_u^{\downarrow}$ and $sp_u^{\uparrow}$ be the shortest path from **r** to u and from **t** to u, respectively, with respect to the lengths l_a. That is,

$$sp_u^{\downarrow} = \begin{cases} 0, & \text{if } u = \mathbf{r}, \\ \min\{l_a + sp_v^{\downarrow} : a = (v,u) \in in(u)\}, & \text{otherwise} \end{cases}$$

and

$$sp_u^{\uparrow} = \begin{cases} 0, & \text{if } u = \mathbf{t}, \\ \min\{l_a + sp_v^{\uparrow} : a = (u,v) \in out(u)\}, & \text{otherwise.} \end{cases}$$

This state information can be computed in linear time (in the size of the given MDD).

Then we delete an arc $a = (u,v)$ with $u \in L_j$ and $j \in J$ whenever every path through a is longer than b; that is,

$$sp_u^{\downarrow} + f_j(d(a)) + sp_v^{\uparrow} > b.$$

This inequality propagator achieves MDD consistency as an arc e is always removed unless there exists a feasible solution to the inequality that supports it [4].

9.4.3 Two-Sided Inequality Constraints

The inequality propagator of Section 9.4.2 can be extended to equalities [84] and two-sided inequalities [94], by storing all path lengths instead of only the shortest and/or longest paths.

Suppose we are given an inequality constraint $L \leq \sum_{j\in J} f_j(x_j) \leq U$, where L and U are numbers such that $L \leq U$. For a node u in a given MDD M, let $s^{\downarrow}(u)$ be the set of all path lengths from $\mathbf{r}$ to u, and $s^{\uparrow}(u)$ the set of all path lengths from u to $\mathbf{t}$. We define $s^{\downarrow}(\mathbf{r}) = \varnothing$ and recursively compute

$$s^{\downarrow}(v) = \begin{cases} \cup_{a'=(u,v)\in in(v)} \{d(a') + w \mid w \in s^{\downarrow}(u)\} & \text{if } u \in L_j \text{ and } j \in J, \\ \cup_{a'=(u,v)\in in(v)} s^{\downarrow}(u) & \text{otherwise.} \end{cases}$$

We delete an arc $a = (u,v)$ if

$$w + d(a) + w' \notin [L,U], \ \text{ for all } w \in s^{\downarrow}(u), w' \in s^{\uparrow}(v). \tag{9.2}$$

Because rule (9.2) for deleting an arc is both a necessary and sufficient condition for the arc to be supported by a feasible solution, one top-down and bottom-up pass suffices to achieve MDD consistency. Observing that the states can be computed in pseudo-polynomial time, by Theorem 9.1 this algorithm runs in pseudo-polynomial time (see also [84]).

9.4.4 ALLDIFFERENT *Constraint*

The constraint ALLDIFFERENT$(\{x_j \mid j \in J)$ requires that the variables x_j for $j \in J$ take distinct values. This constraint was introduced before, and in Section 4.7.1 we presented a filtering procedure to propagate this constraint, based on the states $All_u^{\downarrow}$, $Some_u^{\downarrow}$, $All_u^{\uparrow}$, and $Some_u^{\uparrow}$ (see Lemma 4.2 and Lemma 4.3). The filtering rules presented in those lemmas are not sufficient to establish MDD consistency, however, as illustrated in the following example:

Example 9.9. Consider the constraint ALLDIFFERENT(x_1,x_2,x_3,x_4) with variable domains $x_1 \in \{a,b\}$, $x_2 \in \{a,b,c,d\}$, $x_3 \in \{a,b,c,d\}$, $x_4 \in \{a,b\}$, together with an MDD of width one (ordered x_1,x_2,x_3,x_4) with parallel arcs labeled $D(x_i)$ between layers L_i and L_{i+1} for $i = 1,2,3,4$. Arcs labeled a and b for x_2 and x_3 do not participate in a feasible solution, but the rules in Lemma 4.2 and 4.3 do not detect this.

In fact, we have the following result:

Theorem 9.3 ([4]). *Establishing MDD consistency for the* ALLDIFFERENT *constraint is NP-hard.*

Proof. We reduce the Hamiltonian path problem to enforcing MDD consistency on a particular MDD. Consider a walk of length n (i.e., a path in which vertex repetition is allowed) in a given graph $G = (V,E)$ with n vertices. Let variable x_i represent the i-th vertex visited in the walk. Observe that $x_1,x_2,\ldots,x_n$ is a Hamiltonian path if and only if x_i and x_{i+1} are adjacent, for $i = 1,\ldots,n-1$ and ALLDIFFERENT$(x_1,x_2,\ldots,x_n)$ is satisfied.

We construct a polynomial-sized MDD M, following the variable ordering $x_1,x_2,\ldots,x_n$, as follows: We define $L_1 = \mathbf{r}$, $L_i = \{v_{i,1},v_{i,2},\ldots,v_{i,n}\}$ for $i = 2,\ldots,n$ and $L_{n+1} = \mathbf{t}$. From $\mathbf{r}$ we define arcs $(\mathbf{r},v_{2,j})$ with label j for all $j \in V$. For layers L_i ($i = 2,\ldots,n$), we define arcs $(v_{i,j},v_{i+1,k})$ with label k for all edges $(j,k) \in E$. The MDD thus constructed represents all walks of length n in G.

A Hamiltonian path (if one exists) can now be found iteratively as follows:

1: **for** $i = 1,\ldots,n$ **do**
2: Establish MDD consistency for ALLDIFFERENT on M
3: Let $L_{i+1} = \{v_{i+1,j}\}$ for some arc out of L_i with label j

Since MDD consistency guarantees that all arcs in M belong to a solution to ALLDIFFERENT, this procedure greedily constructs a single **r**–**t** path which is Hamiltonian in G. □

9.4.5 AMONG *Constraint*

The AMONG constraint counts the number of variables that are assigned to a value in a given set S, and ensures that this number is between a given lower and upper bound [21]:

Definition 9.1. Let X be a set of variables, let L,U be integer numbers such that $0 \leq L \leq U \leq |X|$, and let $S \subset \cup_{x \in X} D(x)$ be a subset of domain values. Then we define AMONG(X,S,L,U) as

$$L \leq \sum_{x \in X} (x \in S) \leq U.$$

Note that the expression $(x \in S)$ is evaluated as a binary value, i.e., resulting in 1 if $x \in S$ and 0 if $x \notin S$. The AMONG constraint finds application in scheduling and

sequencing problems, for example to indicate that an employee must be assigned a number of workshifts S between a given minimum L and maximum U every week.

We can reduce propagating $\text{AMONG}(X,S,L,U)$ to propagating a two-sided separable inequality constraint,

$$L \leq \sum_{x_i \in X} f_i(x_i) \leq U,$$

where

$$f_i(v) = \begin{cases} 1, & \text{if } v \in S, \\ 0, & \text{otherwise.} \end{cases}$$

Because each $f_i(\cdot) \in \{0,1\}$, the number of distinct path lengths is bounded by $U - L + 1$, and we can compute all states in polynomial time. Therefore, by Theorem 9.1, MDD consistency can be achieved in polynomial time for AMONG constraints. Observe that tractability also follows from Theorem 9.2, since the size of an exact MDD for AMONG can be bounded by $O(n(U-L+1))$.

To improve the efficiency of this propagator in practice, one may choose to only propagate bounds information. That is, we can use the inequality propagator for the pair of inequalities separately, and reason on the shortest and longest path lengths, as in Section 9.4.2.

9.4.6 ELEMENT *Constraint*

A powerful modeling feature of CP is that variables can be used as subscripts for arrays. That is, for a given array of parameters $c_1,\ldots,c_m$, we can define $y = c_x$, where variable y has domain $\{c_1,\ldots,c_m\}$ and variable x has domain $\{1,\ldots,m\}$. The constraint specifies that y is assigned the x-th element in the array. In the CP literature, this relationship is denoted as the constraint $\text{ELEMENT}(x,(c_1,\ldots,c_m),y)$.

Because this is a binary constraint, we can achieve MDD consistency in polynomial time by defining $s^{\downarrow}(u), s^{\uparrow}(u)$ for all nodes u in the MDD, as in the proof of Corollary 9.1. Let variable x correspond to layer L_i and variable y to L_j. Assuming that $i < j$, we delete an arc $a = (u,v)$ when (a) $u \in L_j$ and there exists no $k \in s^{\downarrow}(u)$ such that $d(a) = c_k$, or (b) $u \in L_i$ and $c_{d(a)} \notin s^{\uparrow}(v)$.

9.4.7 Using Conventional Domain Propagators

It is possible to adapt domain propagators to the context of MDD propagation, as proposed by [4]. We first define the *induced domain relaxation* $D^{\times}(M)$ of an MDD M as a tuple of domains $(D_1^{\times}(M),\ldots,D_n^{\times}(M))$ where each $D_i^{\times}(M)$ contains the values corresponding to layer L_i. That is,

$$D_i^{\times}(M) = \{d(a) \mid a = (u,v), u \in L_i\}.$$

As before, we denote by $M|_u$ the subgraph of M defined by all **r**–**t** paths through u. We then apply, for node $u \in L_i$, a conventional domain propagator to the domains

$$D_1^{\times}(M|_u),\ldots,D_{i-1}^{\times}(M|_u),\{d(a) \mid a \in out(u)\},D_{i+1}^{\times}(M|_u),\ldots,D_n^{\times}(M|_u). \quad (9.3)$$

We remove values only from the domain of x_i, that is from $\{d(a) \mid a \in out(u)\}$, and delete the corresponding arcs from M. This can be done for each node in layer L_i and for each layer in turn. Note that the induced domain relaxation $D^{\times}(M|_u)$ can be computed recursively for all nodes u of a given MDD M in polynomial time (in the size of the MDD).

We can strengthen the propagation by recording which values can be deleted from the domains $D_j^{\times}(M|_u)$ for $j \neq i$ when (9.3) is propagated [4]. If v can be deleted from $D_j^{\times}(M|_u)$, we place the 'nogood' $x_j \neq v$ on each arc in $out(u)$. Then we recursively move the nogoods on level i toward level j. If $j > i$, for example, we propagate (9.3) for each node on level i and then note which nodes on level $i+1$ have the property that all incoming arcs have the nogood $x_j \neq v$. These nodes propagate the nogood to all their outgoing arcs in turn, and so forth until level j is reached, where all arcs with nogood $x_j \neq v$ and label v are deleted.

9.5 Experimental Results

We next provide detailed experimental evidence to support the claim that MDD-based constraint programming can be a viable alternative to constraint programming based on the domain store. The results in this section are obtained with the MDD-based CP solver described in [93], which is implemented in C++. The solver does not propagate the constraints until a fixed point is reached. Instead, by default we

allocate one bottom-up and top-down pass to each constraint. The bottom-up pass is used to compute the state information $s^{\uparrow}$. The top-down pass processes the MDD a layer at a time, in which we first compute $s^{\downarrow}$, then refine the nodes in the layer, and finally apply the propagator conditions based on $s^{\uparrow}$ and $s^{\downarrow}$. Our outer search procedure is implemented using a priority queue, in which the search nodes are inserted with a specific weight. This allows us to easily encode depth-first search or best-first search procedures. Each search tree node contains a copy of the MDD of its parent, together with the associated branching decision.

All the experiments are performed using a 2.33 GHz Intel Xeon machine with 8 GB memory. For comparison reasons, the solver applies a depth-first search, using a static lexicographic-first variable selection heuristic, and a minimum-value-first value selection heuristic. We vary the maximum width of the MDD, while keeping all other settings the same.

Multiple AMONG Constraints

We first present experiments on problems consisting of multiple AMONG constraints. Each instance contains 50 (binary) variables, and each AMONG constraint consists of 5 variables chosen at random, from a normal distribution with a uniform-random mean (from [1..50]) and standard deviation $\sigma = 2.5$, modulo 50. As a result, for these AMONG constraints the variable indices are near-consecutive, a pattern encountered in many practical situations. Each AMONG has a fixed lower bound of 2 and upper bound of 3, specifying the number of variables that can take value 1. In our experiments we vary the number of AMONG constraints (from 5 to 200, by steps of 5) in each instance, and we generate 100 instances for each number. We note that these instances exhibit a sharp feasibility phase transition, with a corresponding computational hardness peak, as the number of constraints increases. We have experimented with several other parameter settings, and we note that the reported results are representative for the other parameter settings; see [93] for more details.

In Fig. 9.3, we provide a scatter plot of the running times for width 1 versus width 4, 8, and 16, for all instances. Note that this is a log–log plot. Points on the diagonal represent instances for which the running times, respectively number of backtracks, are equal. For points below the diagonal, width 4, 8, or 16 has a smaller search tree, respectively is faster, than width 1, and the opposite holds for points above the diagonal.

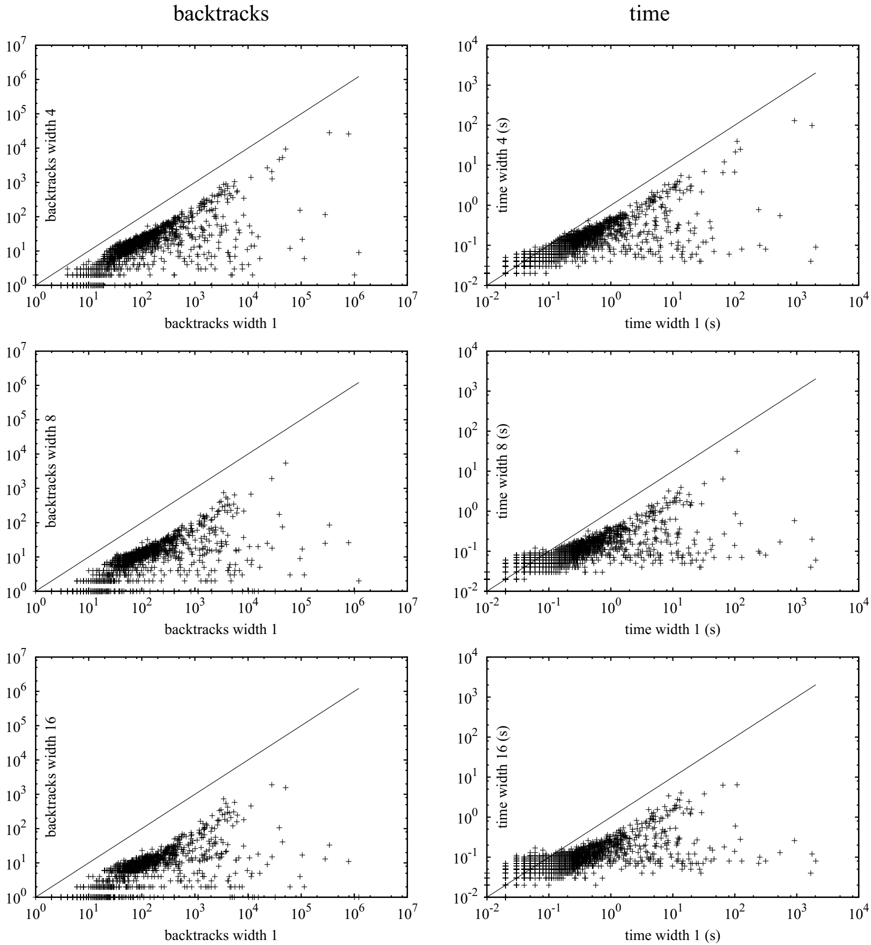


Fig. 9.3 Scatter plots comparing width 1 versus width 4, 8, and 16 (from top to bottom) in terms of backtracks (left) and computation time in seconds (right) on multiple AMONG problems.

We can observe that width 4 already consistently outperforms the domain propagation, in some cases by up to six orders of magnitude in terms of search tree size (backtracks), and up to four orders of magnitude in terms of computation time. For width 8, this behavior is even more consistent, and for width 16, all instances can be solved in under 10 seconds, while the domain propagation needs hundreds or thousands of seconds for several of these instances.

Table 9.1 The effect of the MDD width on time in seconds (CPU) and backtracks (BT) when finding one feasible solution on nurse rostering instances.

Instance		*Width 1*		*Width 2*		*Width 4*		*Width 8*		*Width 16*		*Width 32*		*Width 64*	
	size	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
C-I	40	61,225	55.63	22,443	28.67	8,138	12.64	1,596	3.84	6	0.07	3	0.09	2	0.10
	50	62,700	88.42	20,992	48.82	3,271	12.04	345	2.76	4	0.08	3	0.13	3	0.16
	60	111,024	196.94	38,512	117.66	3,621	19.92	610	6.89	12	0.24	8	0.29	5	0.34
	70	174,417	375.70	64,410	243.75	5,182	37.05	889	12.44	43	0.80	13	0.59	14	0.90
	80	175,175	442.29	64,969	298.74	5,025	44.63	893	15.70	46	1.17	11	0.72	12	1.01
C-II	40	179,743	173.45	60,121	79.44	17,923	32.59	3,287	7.27	4	0.07	4	0.07	5	0.11
	50	179,743	253.55	73,942	166.99	9,663	38.25	2,556	18.72	4	0.09	3	0.12	3	0.18
	60	179,743	329.72	74,332	223.13	8,761	49.66	1,572	16.82	3	0.13	3	0.18	2	0.24
	70	179,743	391.29	74,332	279.63	8,746	64.80	1,569	22.35	4	0.18	2	0.24	2	0.34
	80	179,743	459.01	74,331	339.57	8,747	80.62	1,577	28.13	3	0.24	2	0.32	2	0.45
C-III	40	91,141	84.43	29,781	38.41	5,148	9.11	4,491	9.26	680	1.23	7	0.18	6	0.13
	50	95,484	136.36	32,471	75.59	2,260	9.51	452	3.86	19	0.43	7	0.24	3	0.20
	60	95,509	173.08	32,963	102.30	2,226	13.32	467	5.47	16	0.50	6	0.28	3	0.24
	70	856,470	1,986.15	420,296	1,382.86	37,564	186.94	5,978	58.12	1,826	20.00	87	3.12	38	2.29
	80	882,640	2,391.01	423,053	1,752.07	33,379	235.17	4,236	65.05	680	14.97	55	3.27	32	2.77

Nurse Rostering Instances

We next conduct experiments on a set of instances inspired by nurse rostering problems, taken from [153]. The instances are of three different classes, and combine constraints on the minimum and maximum number of working days for sequences of consecutive days of given lengths. That is, class *C-I* demands to work at most 6 out of each 8 consecutive days (max6/8) and at least 22 out of every 30 consecutive days (min22/30). For class *C-II* these numbers are max6/9 and min20/30, and for class *C-III* these numbers are max7/9 and min22/30. In addition, all classes require to work between 4 and 5 days per calendar week. The planning horizon ranges from 40 to 80 days.

The results are presented in Table 9.1. We report the total number of backtracks upon failure (BT) and computation time in seconds (CPU) needed by our MDD solver for finding a first feasible solution, using widths 1, 2, 4, 8, 16, 32, and 64. Again, the MDD of width 1 corresponds to variable domains. For all problem classes we observe a nearly monotonically decreasing sequence of backtracks and solution time as we increase the width up to 64. Furthermore, the rate of decrease appears to be exponential in many cases, and again higher widths can yield savings of several orders of magnitude. A typical result (the instance *C-III* on 60 days) shows that, where an MDD of width 1 requires 95,509 backtracks and 173.08 seconds of computation time, an MDD of width 32 only requires 6 backtracks and 0.28 seconds of computation time to find a first feasible solution.

Chapter 10
MDD Propagation for SEQUENCE Constraints

Abstract In this chapter we present a detailed study of MDD propagation for the SEQUENCE constraint. This constraint can be applied to combinatorial problems such as employee rostering and car manufacturing. It will serve to illustrate the main challenges when studying MDD propagation for a new constraint type: Tractability, design of the propagation algorithm, and practical efficiency. In particular, we show in this chapter that establishing MDD consistency for SEQUENCE is NP-hard, but fixed-parameter tractable. Furthermore, we present an MDD propagation algorithm that may not establish MDD consistency, but is highly effective in practice when compared to conventional domain propagation.

10.1 Introduction

In the previous chapter we discussed how constraint programming systems can be extended to operate on MDD relaxations. In particular, we saw how the constraint propagation process can be improved if we use an MDD store instead of the conventional domain store. For certain constraint types, however, it is not straightforward to develop effective MDD propagation algorithms. We consider one such constraint type in detail in this chapter, the SEQUENCE constraint.

In order to define the SEQUENCE constraint, we first recall the definition of the AMONG constraint from Section 9.4.5. For a set of variables X, a set of domain values S, and integers $0 \leq L \leq U \leq |X|$, the constraint AMONG(X,S,L,U) specifies that between L and U variables in X are assigned an element from the set S. The

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_10

Table 10.1 Nurse rostering problem specification. Variable set X represents the shifts to be assigned over a sequence of days. The possible shifts are day (D), evening (E), night (N), and day off (O).

Requirement	SEQUENCE(X,S,q,L,U)
At least 20 work shifts every 28 days:	SEQUENCE$(X,\{D,E,N\},28,20,28)$
At least 4 off-days every 14 days:	SEQUENCE$(X,\{O\},14,4,14)$
Between 1 and 4 night shifts every 14 days:	SEQUENCE$(X,\{N\},14,1,4)$
Between 4 and 8 evening shifts every 14 days:	SEQUENCE$(X,\{E\},14,4,8)$
Night shifts cannot appear on consecutive days:	SEQUENCE$(X,\{N\},2,0,1)$
Between 2 and 4 evening/night shifts every 7 days:	SEQUENCE$(X,\{E,N\},7,2,4)$
At most 6 work shifts every 7 days:	SEQUENCE$(X,\{D,E,N\},7,0,6)$

SEQUENCE constraint is the conjunction of a given AMONG constraint applied to every subsequence of length q over a sequence of n variables [21]:

Definition 10.1. Let X be an ordered set of n variables, q,L,U integer numbers such that $0 \leq q \leq n$, $0 \leq L \leq U \leq q$, and $S \subset \cup_{x \in X} D(x)$ a subset of domain values. Then

$$\text{SEQUENCE}(X,S,q,L,U) = \bigwedge_{i=1}^{n-q+1} \text{AMONG}(s_i,S,L,U),$$

where s_i represents the subsequence $x_i,\ldots,x_{i+q-1}$.

The SEQUENCE constraint can be applied in several practical combinatorial problems such as car manufacturing and employee rostering. An illustration is given in the following example:

Example 10.1. Consider a rostering problem in which we need to design a schedule for a nurse over a given horizon of n days. On each day, a nurse can either work a day shift (D), evening shift (E), night shift (N), or have a day off (O). We introduce a variable x_i for each day $i = 1,\ldots,n$, with domain $D(x_i) = \{O,D,E,N\}$ representing the shift on that day. The schedule needs to obey certain requirements, which can all be modeled using SEQUENCE constraints, as listed in Table 10.1.

When modeling a given problem, we have the choice of using separate AMONG constraints or single SEQUENCE constraints. It can be shown that achieving domain consistency on the SEQUENCE constraint is stronger than achieving domain consistency on the decomposition into AMONG constraints. For many practical applications the additional strength from the SEQUENCE constraint is crucial to find

a solution in reasonable time [134, 153]. It is known that conventional domain consistency can be achieved for SEQUENCE in polynomial time [152, 153, 36, 113, 55]. We also know from Section 9.4.5 that MDD consistency can be achieved for the AMONG constraint in polynomial time [94]. The question we address in this chapter is how we can handle SEQUENCE constraints in the context of MDD-based constraint programming.

We first show that establishing MDD consistency on the SEQUENCE constraint is NP-hard. This is an interesting result from the perspective of MDD-based constraint programming. Namely, of all global constraints, the SEQUENCE constraint has perhaps the most suitable combinatorial structure for an MDD approach; it has a prescribed variable ordering, it combines subconstraints on contiguous variables, and existing approaches can handle this constraint fully by using bounds reasoning only.

We then show that establishing MDD consistency on the SEQUENCE constraint is fixed parameter tractable with respect to the lengths of the subsequences (the AMONG constraints), provided that the MDD follows the order of the SEQUENCE constraint. The proof is constructive, and follows from the generic intersection-based algorithm to filter one MDD with another.

The third contribution is a partial MDD propagation algorithm for SEQUENCE, that does not necessarily establish MDD consistency. It relies on the decomposition of SEQUENCE into ‘cumulative sums’, and an extension of MDD filtering to the information that is stored at its nodes.

Lastly, we provide an experimental evaluation of our partial MDD propagation algorithm. We evaluate the strength of the algorithm for MDDs of various maximum widths, and compare the performance with existing domain propagators for SEQUENCE. We also compare our algorithm with the known MDD approach that uses the natural decomposition of SEQUENCE into AMONG constraints [94]. Our experiments demonstrate that MDD propagation can outperform domain propagation for SEQUENCE by reducing the search tree size, and solving time, by several orders of magnitude. Similar results are observed with respect to MDD propagation of AMONG constraints. These results thus provide further evidence for the power of MDD propagation in the context of constraint programming.

10.2 MDD Consistency for SEQUENCE Is NP-Hard

A challenge in determining whether a global constraint can be made MDD consistent in polynomial time is that this must be guaranteed for *any* given MDD. That is, in addition to the combinatorics of the global constraint itself, the shape of the MDD adds another layer of complexity to establishing MDD consistency. For proving NP-hardness, a particular difficulty is making sure that, in the reduction, the MDD remains of polynomial size. One example was given in Section 9.4.4, where we discussed that establishing MDD consistency for the ALLDIFFERENT constraint is NP-hard. We next consider this question for the SEQUENCE constraint, and prove the following result:

Theorem 10.1. *Establishing MDD consistency for* SEQUENCE *on an arbitrary MDD is NP-hard even if the MDD follows the variable ordering of the* SEQUENCE *constraint.*

Proof. The proof is by reduction from 3-SAT, a classical NP-complete problem [71]. We will show that an instance of 3-SAT is satisfied if and only if a particular SEQUENCE constraint on a particular MDD M of polynomial size has a solution. Therefore, establishing MDD consistency for SEQUENCE on an arbitrary MDD is at least as hard as 3-SAT. In this proof, we will use $\ell(u)$ to indicate the layer index of a node u in M, i.e., $u \in L_{\ell(u)}$.

Consider a 3-SAT instance on n variables $x_1,\dots,x_n$, consisting of m clauses $c_1,\dots,c_m$. We first construct an MDD that represents the basic structure of the 3-SAT formula (see Example 10.2 after this proof for an illustration). We introduce binary variables $y_{i,j}$ and $\overline{y}_{i,j}$ representing the literals x_j and $\overline{x}_j$ per clause c_i, for $i = 1,\dots,m$ and $j = 1,\dots,n$ (x_j and $\overline{x}_j$ may or may not exist in c_i). We order these variables as a sequence Y, first by the index of the clauses, then by the index of the variables, and then by $y_{i,j},\overline{y}_{i,j}$ for clause c_i and variable x_j. That is, we have $Y = y_{1,1},\overline{y}_{1,1},\ y_{1,2},\overline{y}_{1,2},\dots,y_{1,n},\overline{y}_{1,n},\ \dots,\ y_{m,1},\overline{y}_{m,1},\ \dots,y_{m,n},\overline{y}_{m,n}$. We construct an MDD M as a layered graph, where the k-th layer corresponds to the k-th variable in the sequence Y.

A clause c_i is represented by $2n$ consecutive layers corresponding to $y_{i,1},\dots,\overline{y}_{i,n}$. In such part of the MDD, we identify precisely those paths that lead to a solution satisfying the clause. The basis for this is a 'diamond' structure for each pair of literals $(y_{i,j},\overline{y}_{i,j})$, that assigns either $(0,1)$ or $(1,0)$ to this pair. If a variable does not appear in a clause, we represent it using such a diamond in the part of the MDD

representing that clause, thus ensuring that the variable can take any assignment with respect to this clause. For the variables that do appear in the clause, we will explicitly list out all allowed combinations.

More precisely, for clause c_i, we first define a local root node r_i representing layer $\ell(y_{i,1})$, and we set $tag(r_i)$ = 'unsat'. For each node u in layer $\ell(y_{i,j})$ (for $j = 1,\ldots,n$), we do the following: If variable x_j does not appear in c_i, or if $tag(u)$ is 'sat', we create two nodes v, v' in $\ell(\overline{y}_{i,j})$, one single node w in $\ell(y_{i,j+1})$, and arcs (u,v) with label 1, (u,v') with label 0, (v,w) with label 0, and (v',w) with label 1. This corresponds to the 'diamond' structure. We set $tag(w) = tag(u)$. Otherwise (i.e., $tag(u)$ is 'unsat' and $y_{i,j}$ appears in c_i), we create two nodes v, v' in $\ell(\overline{y}_{i,j})$, two nodes w, w' in $\ell(y_{i,j+1})$, and arcs (u,v) with label 1, (u,v') with label 0, (v,w) with label 0, and (v',w') with label 1. If c_i contains as literal $y_{i,j}$, we set $tag(w)$ = 'sat' and $tag(w')$ = 'unsat'. Otherwise (c_i contains $\overline{y}_{i,j}$), we set $tag(w)$ = 'unsat' and $tag(w')$ = 'sat'.

This procedure will be initialized by a single root node r representing $\ell(y_{11})$. We iteratively append the MDDs of two consecutive clauses c_i and c_{i+1} by merging the nodes in the last layer of c_i that are marked 'sat' into a single node, and let this node be the local root for c_{i+1}. We finalize the procedure by merging all nodes in the last layer that are marked 'sat' into the single terminal node t. By construction, we ensure that only one of y_{ij} and $\overline{y}_{ij}$ can be set to 1. Furthermore, the variable assignment corresponding to each path between layers $\ell(y_{i,1})$ and $\ell(y_{i+1,1})$ will satisfy clause c_i, and exactly n literals are chosen accordingly on each such path.

We next need to ensure that, for a feasible path in the MDD, each variable x_j will correspond to the same literal $y_{i,j}$ or $\overline{y}_{i,j}$ in each clause c_i. To this end, we impose the constraint

$$\text{SEQUENCE}(Y, S = \{1\}, q = 2n, L = n, U = n) \tag{10.1}$$

on the MDD M described above. If the subsequence of length $2n$ starts from a positive literal $y_{i,j}$, by definition there are exactly n variables that take value 1. If the subsequence starts from a negative literal $\overline{y}_{i,j}$ instead, the last variable in the sequence corresponds to the value x_j in the next clause c_{i+1}, i.e., $y_{i+1,j}$. Observe that all variables except for the first and the last in this sequence will take value 1 already $n-1$ times. Therefore, of the first and the last variable in the sequence (which represent x_j and its complement $\overline{x}_j$ in any order), only one can take the value 1. That is, x_j must take the same value in clause c_i and c_{i+1}. Since this holds for all subsequences, all variables x_j must take the same value in all clauses.

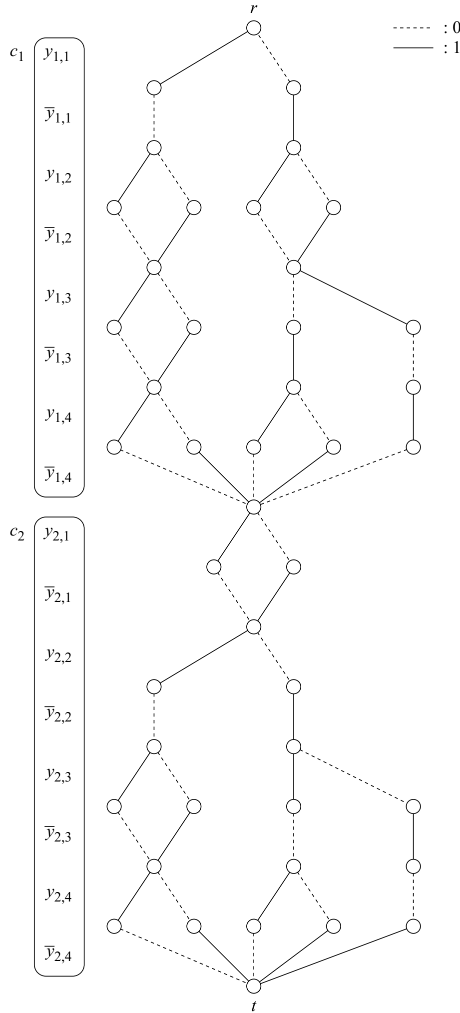


Fig. 10.1 The MDD corresponding to Example 10.2.

The MDD M contains $2mn+1$ layers, while each layer contains at most six nodes. Therefore, it is of polynomial size (in the size of the 3-SAT instance), and the overall construction needs polynomial time. □

Example 10.2. Consider the 3-SAT instance on four Boolean variables x_1, x_2, x_3, x_4 with clauses $c_1 = (x_1 \vee \overline{x}_3 \vee x_4)$ and $c_2 = (x_2 \vee x_3 \vee \overline{x}_4)$. The corresponding MDD used in the reduction is given in Fig. 10.1.

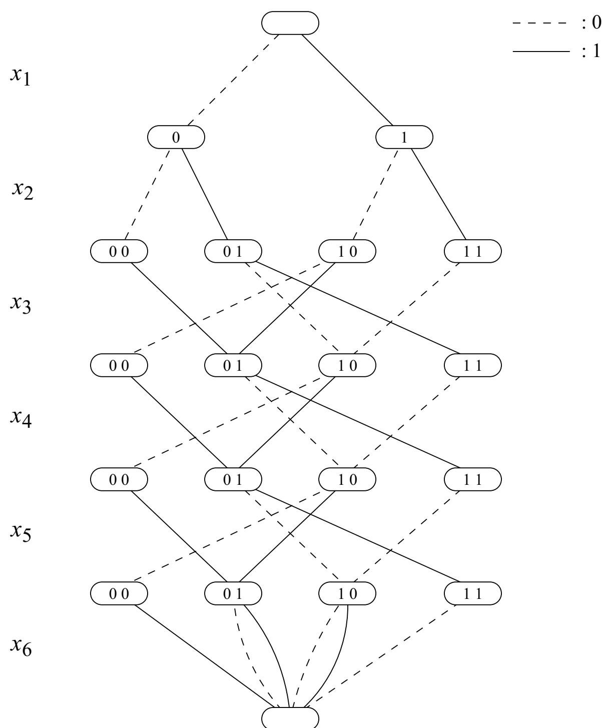


Fig. 10.2 The exact MDD for the SEQUENCE constraint of Example 10.3.

10.3 MDD Consistency for SEQUENCE Is Fixed Parameter Tractable

In this section we show that establishing MDD consistency for SEQUENCE on an arbitrary MDD is fixed parameter tractable, with respect to the length q of the subsequences. It was already shown in [152, 153] that an exact MDD for the SEQUENCE constraint exists with $O(n2^q)$ nodes (i.e., the 'unfolded' automaton of the REGULAR constraint), as illustrated in the next example.

Example 10.3. Consider the constraint SEQUENCE$(X, S = \{1\}, q = 3, L = 1, U = 2)$ where $X = \{x_1, x_2, \ldots, x_6\}$ is an ordered set of binary variables. The corresponding exact MDD, following the order of X, is presented in Fig. 10.2. For convenience, each node in the MDD is labeled with the last $q-1$ labels that represent the subsequence up to that node (starting $q-1$ layers up). For example, the second node in the third layer represents decisions $x_1 = 0$ and $x_2 = 1$, corresponding to

subsequence 01. To construct the next layer, we append either a 0 or a 1 to this subsequence (and remove the first symbol), leading to nodes labeled 10 and 11, respectively. Note that from nodes labeled 00 we must take an arc with label 1, because $L = 1$. Similarly for nodes labeled 11 we must take an arc with label 0, because $U = 2$. After q layers, all possible subsequences have been created (maximally $O(2^{q-1})$), which thus defines the width of the subsequent layers.

In Section 9.3.3 we presented a generic MDD propagation algorithm that establishes MDD consistency of one MDD with respect to another by taking the intersection (Algorithm 8). We can apply that algorithm, and the associated Theorem 9.2, to the SEQUENCE constraint to obtain the following result:

Corollary 10.1. *Let* $C = \text{SEQUENCE}(X,S,q,L,U)$ *be a sequence constraint, where* X *is an ordered sequence of variables, and let* M *be an arbitrary MDD following the variable ordering of* X*. Establishing MDD consistency for* C *on* M *is fixed parameter tractable with respect to parameter* q*.*

Proof. We know from [152, 153] that there exists an exact MDD M' of size $O(n2^{q-1})$ that represents C. Applying Theorem 9.2 gives an MDD-consistency algorithm with time and space complexity $O(|M|\,2^{q-1})$, and the result follows. □

10.4 Partial MDD Filtering for SEQUENCE

In many practical situations the value of q will lead to prohibitively large exact MDDs for establishing MDD consistency, which limits the applicability of Corollary 10.1. Therefore we next explore a more practical partial filtering algorithm that is polynomial also in q.

One immediate approach is to propagate the SEQUENCE constraint in MDDs through its natural decomposition into AMONG constraints, and apply the MDD filtering algorithms for AMONG proposed by [94]. However, it is well known that, for classical constraint propagation based on variable domains, the AMONG decomposition can be substantially improved by a dedicated domain filtering algorithm for SEQUENCE [152, 153, 36, 113]. Therefore, our goal in this section is to provide MDD filtering for SEQUENCE that can be stronger in practice than MDD filtering for the AMONG decomposition, and stronger than domain filtering for SEQUENCE. In what follows, we assume that the MDD at hand respects the ordering of the variables in the SEQUENCE constraint.

10.4.1 Cumulative Sums Encoding

Our proposed algorithm extends the original domain consistency filtering algorithm for SEQUENCE by [152] to MDDs, following the 'cumulative sums' encoding as proposed by [36]. This representation takes the following form: For a sequence of variables $X = x_1, x_2, \ldots, x_n$, and a constraint SEQUENCE(X, S, q, L, U), we first introduce variables $y_0, y_1, \ldots, y_n$, with respective initial domains $D(y_i) = [0, i]$ for $i = 1, \ldots, n$. These variables represent the cumulative sums of X, i.e., y_i represents $\sum_{j=1}^{i} (x_j \in S)$ for $i = 1, \ldots, n$. We now rewrite the SEQUENCE constraint as the following system of constraints:

$$y_i = y_{i-1} + \delta_S(x_i) \quad \forall i \in \{1, \ldots, n\}, \tag{10.2}$$

$$y_{i+q} - y_i \geq l \quad \forall i \in \{0, \ldots, n-q\}, \tag{10.3}$$

$$y_{i+q} - y_i \leq u \quad \forall i \in \{0, \ldots, n-q\}, \tag{10.4}$$

where $\delta_S : X \to \{0, 1\}$ is the indicator function for the set S, i.e., $\delta_S(x) = 1$ if $x \in S$ and $\delta_S(x) = 0$ if $x \notin S$. [36] show that establishing singleton bounds consistency on this system suffices to establish domain consistency for the original SEQUENCE constraint.

In order to apply similar reasoning in the context of MDDs, the crucial observation is that the domains of the variables $y_0, \ldots, y_n$ can be naturally represented at the *nodes* of the MDD. In other words, a node v in layer L_i represents the domain of y_{i-1}, restricted to the solution space formed by all **r**–**t** paths containing v. Let us denote this information for each node v explicitly as the interval $[\text{lb}(v), \text{ub}(v)]$, and we will refer to it as the 'node domain' of v. Following the approach of [94], we can compute this information in linear time by one top-down pass, by using equation (10.2), as follows:

$$\begin{aligned} \text{lb}(v) &= \min_{(u,v) \in in(v)} \{\text{lb}(u) + \delta_S(d(u,v))\}, \\ \text{ub}(v) &= \max_{(u,v) \in in(v)} \{\text{ub}(u) + \delta_S(d(u,v))\}, \end{aligned} \tag{10.5}$$

for all nodes $v \neq r$, while $[\text{lb}(r), \text{ub}(r)] = [0, 0]$.

As the individual AMONG constraints are now posted as $y_{i+q} - y_i \geq l$ and $y_{i+q} - y_i \leq u$, we also need to compute for a node v in layer L_{i+1} all its ancestors from layer L_i. This can be done by maintaining a vector $\mathscr{A}_v$ of length $q+1$ for each node v, where $\mathscr{A}_v[i]$ represents the set of ancestor nodes of v at the i-th layer above

v, for $i = 0, \ldots, q$. We initialize $\mathscr{A}_r = [\{r\}, \emptyset, \ldots, \emptyset]$, and apply the recursion

$$\mathscr{A}_v[i] = \cup_{(u,v)\in in(v)} \mathscr{A}_u[i-1] \quad \text{for } i = 1, 2, \ldots, q,$$
$$\mathscr{A}_v[0] = \{v\}.$$

The resulting top-down pass itself takes linear time (in the size of the MDD), while a direct implementation of the recursive step for each node takes $O(q \cdot (\omega(M))^2)$ operations for an MDD M. Now, the relevant ancestor nodes for a node v in layer L_{i+q} are stored in $\mathscr{A}_v[q]$, a subset of layer L_i. We similarly compute all descendant nodes of v in a vector $\mathscr{D}_v$ of length $q+1$, such that $\mathscr{D}_v[i]$ contains all descendants of v in the i-th layer below v, for $i = 0, 1, \ldots, q$. We initialize $\mathscr{D}_t = [\{t\}, \emptyset, \ldots, \emptyset]$.

However, for our purposes we only need to maintain the minimum and maximum value of the union of the domains of $\mathscr{A}_v$, resp. $\mathscr{D}_v$, because constraints (10.3) and (10.4) are inequalities; see the application of $\mathscr{A}_v$ and $\mathscr{D}_v$ in rules (10.8) below. This makes the recursive step more efficient, now taking $O(q\omega(M))$ operations per node.

Alternatively, we can approximate this information by only maintaining a minimum and maximum node domain value for each *layer*, instead of a list of ancestor layers. This will compromise the filtering, but may be more efficient in practice, as it only requires to maintain two integers per layer.

10.4.2 Processing the Constraints

We next process each of the constraints (10.2), (10.3), and (10.4) in turn to remove provably inconsistent arcs, while at the same time we filter the node information.

Starting with the ternary constraints of type (10.2), we remove an arc (u, v) if $\text{lb}(u) + \delta_S(d(u, v)) > ub(v)$. Updating $[\text{lb}(v), \text{ub}(v)]$ for a node v is done similarly to the rules (10.5) above:

$$\begin{aligned} \text{lb}(v) &= \max\left\{\text{lb}(v), \min_{(u,v)\in in(v)} \{\text{lb}(u) + \delta_S(d(u,v))\}\right\}, \\ \text{ub}(v) &= \min\left\{\text{ub}(v), \min_{(u,v)\in in(v)} \{\text{ub}(u) + \delta_S(d(u,v))\}\right\}. \end{aligned} \tag{10.6}$$

In fact, the resulting algorithm is a special case of the MDD consistency equality propagator of [84], and we thus inherit the MDD consistency for our ternary constraints.

Next, we process the constraints (10.3) and (10.4) for a node v in layer L_{i+1} ($i = 0,\ldots,n$). Recall that the relevant ancestors from L_{i+1-q} are $\mathscr{A}_v[q]$, while its relevant descendants from L_{i+1+q} are $\mathscr{D}_v[q]$. The variable corresponding to node v is y_i, and it participates in four constraints:

$$\begin{aligned} y_i &\geq l + y_{i-q}, \\ y_i &\leq u + y_{i-q}, \\ y_i &\leq y_{i+q} - l, \\ y_i &\geq y_{i+q} - u. \end{aligned} \tag{10.7}$$

Note that we can apply these constraints to filter *only* the node domain $[\mathrm{lb}(v), \mathrm{ub}(v)]$ corresponding to y_i. Namely, the node domains corresponding to the other variables y_{i-q} and y_{i+q} may find support from nodes in layer L_{i+1} other than v. We update $\mathrm{lb}(v)$ and $\mathrm{ub}(v)$ according to equations (10.7):

$$\begin{aligned} \mathrm{lb}(v) &= \max\{\ \mathrm{lb}(v),\ \ l + \min_{u \in \mathscr{A}_v[q]} \mathrm{lb}(u),\ \ \min_{w \in \mathscr{D}_v[q]} \mathrm{lb}(w) - u\ \}, \\ \mathrm{ub}(v) &= \min\{\ \mathrm{ub}(v),\ u + \max_{u \in \mathscr{A}_v[q]} \mathrm{ub}(u),\ \ \max_{w \in \mathscr{D}_v[q]} \mathrm{ub}(w) - l\ \}. \end{aligned} \tag{10.8}$$

The resulting algorithm is a specific instance of the generic MDD-consistent binary constraint propagator presented by [94], and again we inherit the MDD consistency for these constraints. We can process the constraints in linear time (in the size of the MDD) by a top-down and bottom-up pass through the MDD.

Example 10.4. Consider the constraint $\textsc{Sequence}(X, S = \{1\}, q = 3, L = 1, U = 2)$ with the ordered sequence of binary variables $X = \{x_1, x_2, x_3, x_4, x_5\}$. Assume we are given the MDD in Fig. 10.3(a). In Fig. 10.3(b) we show the node domains that result from processing rules (10.5). Figure 10.3(c) shows the resulting MDD after processing the constraints via the rules (10.6) and (10.8). For example, consider the middle node in the fourth layer, corresponding to variable y_3. Let this node be v. It has initial domain $[0,2]$, and $\mathscr{A}_v[q]$ only contains the root node, which has domain $[0,0]$. Since $l = 1$, we can reduce the domain of v to $[1,2]$. We can next consider the arcs into v, and conclude that value 1 in its domain is not supported. This further reduces the domain of v to $[2,2]$, and allows us to eliminate one incoming arc (from the first node of the previous layer).

The resulting MDD in Fig. 10.3(c) reflects all possible deductions that can be made by our partial algorithm. We have not established MDD consistency however, as witnessed by the infeasible path $(1,1,0,0,0)$.

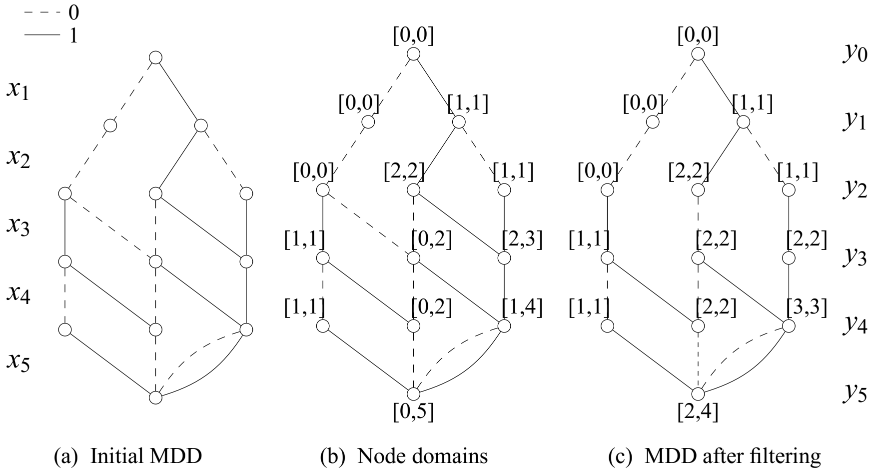


Fig. 10.3 MDD propagation for the constraint SEQUENCE$(X, S = \{1\}, q = 3, L = 1, U = 2)$ of Example 10.4.

The authors in [153] defined the *generalized* SEQUENCE constraint, which extends the SEQUENCE constraint by allowing the AMONG constraints to be specified with different lower and upper bounds, and subsequence lengths:

Definition 10.2. Let X be an ordered set of n variables, k a natural number, $\mathbf{s}$, $\mathbf{l}$, $\mathbf{u}$ vectors of length k such that s_i is a subsequence of X, $l_i, u_i \in \mathbb{N}$, $0 \leq l_i \leq u_i \leq n$ for $i = 1, 2, \ldots, k$, and $S \subset \cup_{x \in X} D(x)$ a subset of domain values. Then

$$\text{GEN-SEQUENCE}(X, S, \mathbf{s}, \mathbf{l}, \mathbf{u}) = \bigwedge_{i=1}^{k} \text{AMONG}(s_i, S, l_i, u_i).$$

Observe that our proposed algorithm can be applied immediately to the GEN-SEQUENCE constraint. The cumulative sums encoding can be adjusted in a straightforward manner to represent these different values.

10.4.3 Formal Analysis

We next investigate the strength of our partial MDD filtering algorithm with respect to other consistency notions for the SEQUENCE constraint. In particular, we formally compare the outcome of our partial MDD filtering algorithm with MDD propagation for the AMONG encoding and with domain propagation for SEQUENCE. We say

that two notions of consistency for a given constraint are *incomparable* if one is not always at least as strong as the other.

First, we recall Theorem 4 from [36].

Theorem 10.2. [36] *Bounds consistency on the cumulative sums encoding is incomparable to bounds consistency on the* AMONG *encoding of* SEQUENCE.

Note that, since all variable domains in the AMONG and cumulative sums encoding are ranges (intervals of integer values), bounds consistency is equivalent to domain consistency for these encodings.

Corollary 10.2. *MDD consistency on the cumulative sums encoding is incomparable to MDD consistency on the* AMONG *encoding of* SEQUENCE.

Proof. We apply the examples from the proof of Theorem 4 in [36]. Consider the constraint $\text{SEQUENCE}(X, S = \{1\}, q = 2, L = 1, U = 2)$ with the ordered sequence of binary variables $X = \{x_1, x_2, x_3, x_4\}$ and domains $D(x_i) = \{0, 1\}$ for $i = 1, 2, 4$, and $D(x_3) = \{0\}$. We apply the 'trivial' MDD of width 1 representing the Cartesian product of the variable domains. Establishing MDD consistency on the cumulative sums encoding yields

$$\begin{aligned}&y_0 \in [0,0], y_1 \in [0,1], y_2 \in [1,2], y_3 \in [1,2], y_4 \in [2,3],\\&x_1 \in \{0,1\}, x_2 \in \{0,1\}, x_3 \in \{0\}, x_4 \in \{0,1\}.\end{aligned}$$

Establishing MDD consistency on the AMONG encoding, however, yields

$$x_1 \in \{0,1\}, \mathbf{x_2} \in \{\mathbf{1}\}, x_3 \in \{0\}, \mathbf{x_4} \in \{\mathbf{1}\}.$$

Consider the constraint $\text{SEQUENCE}(X, S = \{1\}, q = 3, L = 1, U = 1)$ with the ordered sequence of binary variables $X = \{x_1, x_2, x_3, x_4\}$ and domains $D(x_i) = \{0, 1\}$ for $i = 2, 3, 4$, and $D(x_1) = \{0\}$. Again, we apply the MDD of width 1 representing the Cartesian product of the variable domains. Establishing MDD consistency on the cumulative sums encoding yields

$$\begin{aligned}&y_0 \in [0,0], y_1 \in [0,0], y_2 \in [0,1], y_3 \in [1,1], y_4 \in [1,1],\\&x_1 \in \{0\}, x_2 \in \{0,1\}, x_3 \in \{0,1\}, \mathbf{x_4} \in \{\mathbf{0}\},\end{aligned}$$

while establishing MDD consistency on the AMONG encoding does not prune any value. ☐

As an additional illustration of Corollary 10.2, consider again Example 10.4 and Fig. 10.3. MDD propagation for the AMONG encoding will eliminate the value $x_4 = 0$ from the infeasible path $(1,1,0,0,0)$, whereas our example showed that MDD propagation for cumulative sums does not detect this.

Theorem 10.3. *MDD consistency on the cumulative sums encoding of* SEQUENCE *is incomparable to domain consistency on* SEQUENCE.

Proof. The first example in the proof of Corollary 10.2 also shows that domain consistency on SEQUENCE can be stronger than MDD consistency on the cumulative sums encoding.

To show the opposite, consider a constraint SEQUENCE$(X, S = \{1\}, q, L, U)$ with a set of binary variables of arbitrary size, arbitrary values q, L, and $U = |X| - 1$. Let M be the MDD defined over X consisting of two disjoint paths from r to t: the arcs on one path all have label 0, while the arcs on the other all have value 1. Since the projection onto the variable domains gives $x \in \{0,1\}$ for all $x \in X$, domain consistency will not deduce infeasibility. However, establishing MDD consistency with respect to M on the cumulative sums encoding will detect this. □

Even though formally our MDD propagation based on cumulative sums is incomparable to domain propagation of SEQUENCE and MDD propagation of AMONG constraints, in the next section we will show that in practice our algorithm can reduce the search space by orders of magnitude compared with these other methods.

10.5 Computational Results

The purpose of our computational results is to evaluate empirically the strength of the partial MDD propagator described in Section 10.4. We perform three main comparisons. First, we want to assess the impact of increasing the maximum width of the MDD on the filtering. Second, we want to compare the MDD propagation with the classical domain propagation for SEQUENCE. In particular, we wish to evaluate the computational overhead of MDD propagation relative to domain propagation, and to what extent MDD propagation can outperform domain propagation. Third, we compare the filtering strength of our MDD propagator for SEQUENCE with the filtering strength of the MDD propagators for the individual AMONG constraints, the latter being the best MDD approach for SEQUENCE so far [94].

We have implemented our MDD propagator for SEQUENCE as a custom global constraint in IBM ILOG CPLEX CP Optimizer 12.4, using the C++ interface. Recall from Section 10.4 that for applying rules (10.8) we can either maintain a minimum and maximum value for the q previous ancestors and descendants of each node, or approximate this by maintaining these values simply for each layer. We evaluated both strategies and found that the latter did reduce the amount of filtering, but nonetheless resulted in much more efficient performance (about twice as fast on average). Hence, the reported results use that implementation.

For the MDD propagator for AMONG, we apply the code of [94]. For the domain propagation, we applied three models. The first uses the domain-consistent propagator for SEQUENCE from [153], running in $O(n^3)$ time. The second uses the domain-consistent propagator for SEQUENCE based on a network flow representation by [113], which runs in $O(n^2)$ time.[1] As a third model, we applied the decomposition into cumulative sums, which uses no explicit global constraint for SEQUENCE. Propagating this decomposition also takes $O(n^2)$ in the worst case, as it considers $O(n)$ variables and constraints while the variable domains contain up to n elements. We note that, for almost all test instances, the cumulative sums encoding established domain consistency. As an additional advantage, the cumulative sums encoding permits a more insightful comparison with our MDD propagator, since both are based on the cumulative sums decomposition.

We note that [36] introduce the 'multiple-SEQUENCE' constraint that represents the conjunction of multiple SEQUENCE constraints on the same set of ordered variables (as in our experimental setup). [119] shows that establishing bounds consistency on such a system is already NP-hard, and presents a domain-consistent propagator that encodes the system as an automaton for the REGULAR constraint. The algorithm runs in $O(nm^q)$ time, where n represents the number of variables, m the number of SEQUENCE constraints, and q the length of the largest subsequence.

In order to compare our algorithms with the multiple-SEQUENCE constraint, we conducted experiments to identify a suitable testbed. We found that instances for which the multiple-SEQUENCE constraint would not run out of memory could be solved instantly by using any domain propagator for the individual SEQUENCE constraints, while creating the multiple-SEQUENCE constraint took substantially more time on average. For instances that were more challenging (as described in the next sections), the multiple-SEQUENCE constraint could not be applied due to

[1] The implementation was shared by Nina Narodytska.

memory issues. We therefore excluded this algorithm from the comparisons in the sections below.

Because single SEQUENCE constraints can be solved in polynomial time, we consider instances with multiple SEQUENCE constraints in our experiments. We assume that these are defined on the same ordered set of variables. To measure the impact of the different propagation methods correctly, all approaches apply the same fixed search strategy, i.e., following the given ordering of the variables, with a lexicographic value ordering heuristic. For each method, we measure the number of backtracks from a failed search state as well as the solving time. All experiments are performed using a 2.33 GHz Intel Xeon machine.

10.5.1 Systems of SEQUENCE *Constraints*

We first consider systems of multiple SEQUENCE constraints that are defined on the same set of variables. We generate instances with $n = 50$ variables each having domain $\{0,1,\ldots,10\}$, and five SEQUENCE constraints. For each SEQUENCE constraint, we set the length of subsequence uniform randomly between $[5, n/2)$ as

$$q = (\texttt{rand()}\%((n/2)-5))+5.$$

Here, `rand()` refers to the standard C++ random number generator, i.e., `rand()`$\%k$ selects a number in the range $[0, k-1]$. Without the minimum length of 5, many of the instances would be very easy to solve by either method. We next define the difference between l and u as $\Delta := (\texttt{rand()}\%q)$, and set

$$l := (\texttt{rand()}\%(q-\Delta)),$$
$$u := l + \Delta.$$

Lastly, we define the set of values S by defining its cardinality as $(\texttt{rand()}\%11)+1$, and then selecting that many values uniformly at random from $\{0,1,\ldots,10\}$. We generated 250 such instances in total.[2]

We solve each instance using the domain consistency propagator for SEQUENCE, the cumulative sums encoding (domain propagation), and the MDD propagator with maximum widths 2, 4, 8, 16, 32, 64, 128. Each method is given a maximum time limit of 1,800 seconds per instance.

[2] All instances are available at `http://www.andrew.cmu.edu/user/vanhoeve/mdd/`.

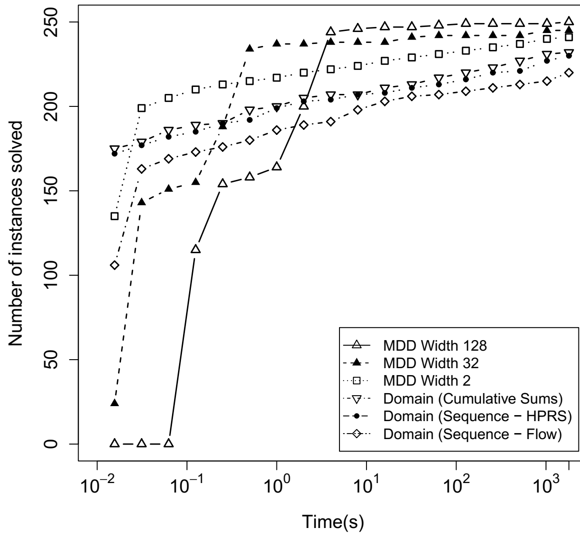
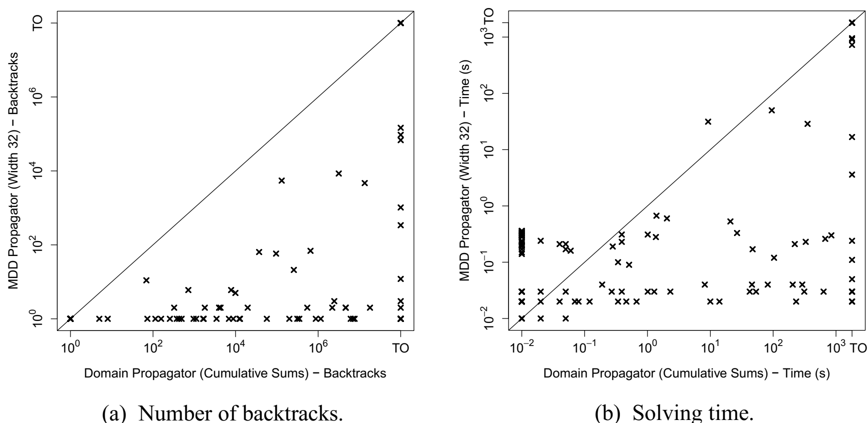


Fig. 10.4 Performance comparison of domain and MDD propagators for the SEQUENCE constraint. Each data point reflects the total number of instances that are solved by a particular method within the corresponding time limit.

We compare the performance of domain propagation and MDD propagation in Fig. 10.4. In this figure, we report for each given time point how many instances could be solved within that time by a specific method. The three domain propagation methods are represented by 'Cumulative Sums' (the cumulative sums decomposition), 'Sequence - HPRS' (the SEQUENCE propagator by [152, 153]), and 'Sequence - Flow' (the flow-based propagator by [113]). Observe that the cumulative sums domain propagation, although not guaranteed to establish domain consistency, outperforms both domain-consistent SEQUENCE propagators. Also, MDD propagation with maximum width 2 can already substantially outperform domain propagation. We can further observe that larger maximum widths require more time for the MDDs to be processed, but in the end it does allow us to solve more instances: maximum MDD width 128 permits to solve all 250 instances within the given time limit, whereas domain propagation can respectively solve 220 (Sequence - Flow), 230 (Sequence - HPRS), and 232 (Cumulative Sums) instances.

To illustrate the difference between domain and MDD propagation in more detail, Fig. 10.5 presents scatter plots comparing domain propagation (cumulative

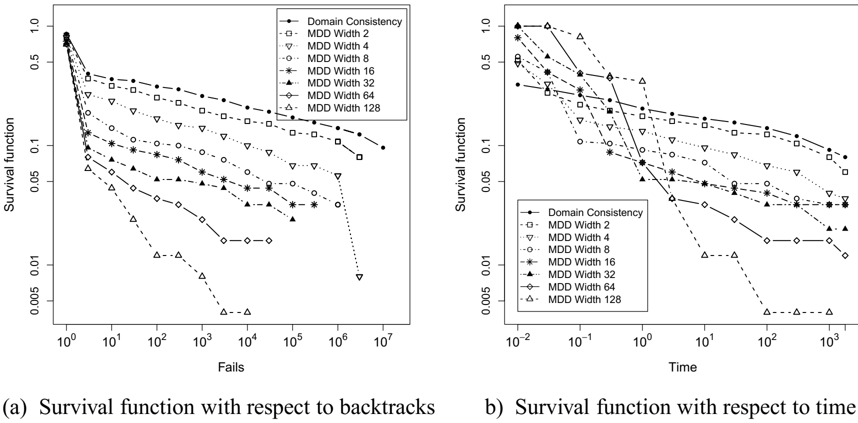


(a) Number of backtracks. (b) Solving time.

Fig. 10.5 Comparing domain and MDD propagation for SEQUENCE constraints. Each data point reflects the number of backtracks (a) resp. solving time in seconds (b) for a specific instance, when solved with the best domain propagator (cumulative sums encoding) and the MDD propagator with maximum width 32. Instances for which either method needed 0 backtracks (a) or less than 0.01 seconds (b) are excluded. Here, TO stands for 'timeout' and represents that the specific instance could not be solved within 1,800 s (b). In (a), these instances are labeled separately by TO (at tick-mark 10^8); note that the reported number of backtracks after 1,800 seconds may be much less than 10^8 for these instances. All reported instances with fewer than 10^8 backtracks were solved within the time limit.

sums) with MDD propagation (maximum width 32). This comparison is particularly meaningful because both propagation methods rely on the cumulative sums representation. For each instance, Fig. 10.5(a) depicts the number of backtracks while Fig. 10.5(b) depicts the solving time of both methods. The instances that were not solved within the time limit are collected under 'TO' (time out) for that method. Figure 10.5(a) demonstrates that MDD propagation can lead to dramatic search tree reductions, by several orders of magnitude. Naturally, the MDD propagation comes with a computational cost, but Fig. 10.5(b) shows that, for almost all instances (especially the harder ones), the search tree reductions correspond to faster solving times, again often by several orders of magnitude.

We next evaluate the impact of increasing maximum widths of the MDD propagator. In Fig. 10.6, we present for each method the 'survival function' with respect to the number of backtracks (a) and solving time (b). Formally, when applied to combinatorial backtrack search algorithms, the survival function represents the probability of a run taking more than x backtracks [75]. In our case, we approximate this function by taking the proportion of instances that need at least x backtracks



(a) Survival function with respect to backtracks b) Survival function with respect to time

Fig. 10.6 Evaluating the impact of increased width for MDD propagation via survival function plots with respect to search backtracks (a) and solving time (b). Both plots are in log–log scale. Each data point reflects the percentage of instances that require at least that many backtracks (a) resp. seconds (b) to be solved by a particular method.

(Fig. 10.6(a)), respectively seconds (Fig. 10.6(b)). Observe that these are log-log plots. With respect to the search tree size, Fig. 10.6(a) clearly shows the strengthening of the MDD propagation when the maximum width is increased. In particular, the domain propagation reflects the linear behavior over several orders of magnitude that is typical for heavy-tailed runtime distributions. Naturally, similar behavior is present for the MDD propagation, but in a much weaker form for increasing maximum MDD widths. The associated solving times are presented in Fig. 10.6(b). It reflects similar behavior, but also takes into account the initial computational overhead of MDD propagation.

10.5.2 Nurse Rostering Instances

We next consider the nurse rostering problem defined in Example 10.1, which represents a more structured problem class. That is, we define a constraint satisfaction problem on variables x_i ($i = 1,\ldots,n$), with domains $D(x_i) = \{O,D,E,N\}$ representing the shift for the nurse. We impose the eight SEQUENCE constraints modeling the requirements listed in Table 10.1. By the combinatorial nature of this problem, the size of the CP search tree turns out to be largely independent of the

Table 10.2 Comparing domain propagation and the MDD propagation for SEQUENCE on nurse rostering instances. Here, n stands for the number of variables, BT for the number of backtracks, and CPU for solving time in seconds.

	Domain Sequence		*Domain Cumul. Sums*		*MDD Width 1*		*MDD Width 2*		*MDD Width 4*		*MDD Width 8*	
n	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
40	438,059	43.83	438,059	32.26	438,059	54.27	52,443	12.92	439	0.44	0	0.02
60	438,059	78.26	438,059	53.40	438,059	80.36	52,443	18.36	439	0.68	0	0.04
80	438,059	124.81	438,059	71.33	438,059	106.81	52,443	28.58	439	0.94	0	0.06
100	438,059	157.75	438,059	96.27	438,059	135.37	52,443	37.76	439	1.22	0	0.10

length of the time horizon, when a lexicographic search (by increasing day i) is applied. We however do consider instances with various time horizons ($n = 40$, 60, 80, 100), to address potential scaling issues.

The results are presented in Table 10.2. The columns for 'Domain Sequence' show the total number of backtracks (BT) and solving time in seconds (CPU) for the domain-consistent SEQUENCE propagator. Similarly, the columns for 'Domain Cumul. Sums' show this information for the cumulative sums domain propagation. The subsequent columns show these numbers for the MDD propagator, for MDDs of maximum width 1, 2, 4, and 8. Note that propagating an MDD of width 1 corresponds to domain propagation, and indeed the associated number of backtracks is equivalent to the domain propagator of the cumulative sums. As a first observation, a maximum width of 2 already reduces the number of backtracks by a factor of 8.3. For maximum width of 8 the MDD propagation even allows to solve the problem without search. The computation times are correspondingly reduced, e.g., from 157 s (resp. 96 s) for the domain propagators to 0.10 s for the MDD propagator (width 8) for the instance with $n = 100$. Lastly, we can observe that in this case MDD propagation does not suffer from scaling issues when compared with domain propagation.

As a final remark, we also attempted to solve these nurse rostering instances using the SEQUENCE domain propagator of CP Optimizer (`IloSequence`). It was able to solve the instance with $n = 40$ in 1,150 seconds, but none of the other instances were solved within the time limit of 1,800 seconds.

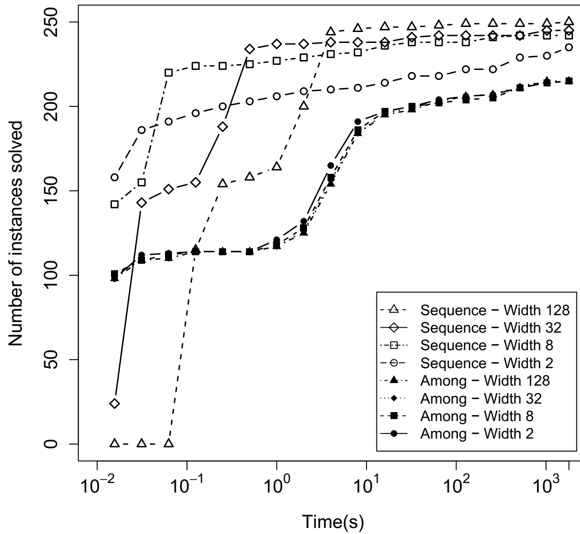


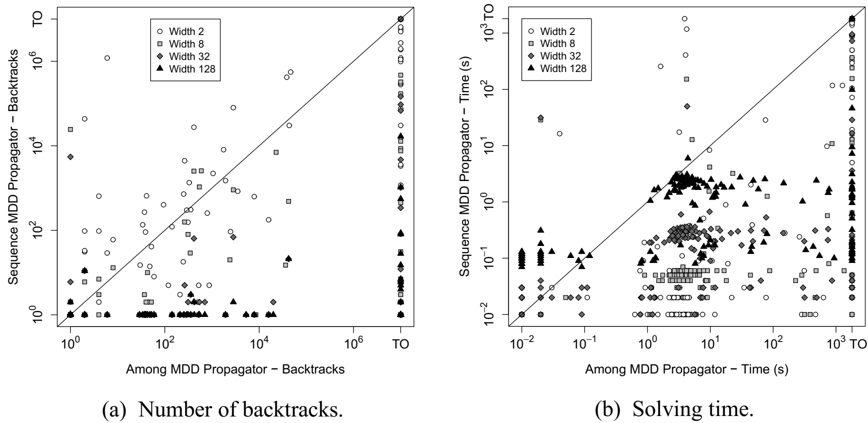
Fig. 10.7 Performance comparison of MDD propagation for SEQUENCE and AMONG for various maximum widths. Each data point reflects the total number of instances that are solved by a particular method within the corresponding time limit.

10.5.3 Comparing MDD Filtering for SEQUENCE *and* AMONG

In our last experiment, we compare our SEQUENCE MDD propagator with the MDD propagator for AMONG constraints by [94]. Our main goal is to determine whether a large MDD is by itself sufficient to solve these problem (irrespective of propagating AMONG or a cumulative sums decomposition), or whether the additional information obtained by our SEQUENCE propagator makes the difference.

We apply both methods, i.e., MDD propagation for SEQUENCE and MDD propagation for AMONG, to the dataset of Section 10.5.1 containing 250 instances. The time limit is again 1,800 seconds, and we run the propagators with maximum MDD widths 2, 8, 32, and 128.

We first compare the performance of the MDD propagators for AMONG and SEQUENCE in Fig. 10.7. The figure depicts the number of instances that can be solved within a given time limit for the various methods. The plot indicates that the AMONG propagators are much weaker than the SEQUENCE propagator, and moreover that larger maximum widths alone do not suffice: using the SEQUENCE



(a) Number of backtracks.

(b) Solving time.

Fig. 10.8 Evaluating MDD propagation for SEQUENCE and AMONG for various maximum widths via scatter plots with respect to search backtracks (a) and solving time (b). Both plots are in log–log scale and follow the same format as Fig. 10.5.

propagator with maximum width 2 outperforms the AMONG propagators for all maximum widths up to 128.

The scatter plot in Fig. 10.8 compares the MDD propagators for AMONG and SEQUENCE in more detail, for widths 2, 8, 32, and 128 (instances that take 0 backtracks, resp. less than 0.01 seconds, for either method are discarded from Fig. 10.8(a), resp. 10.8(b)). For smaller widths, there are several instances that the AMONG propagator can solve faster, but the relative strength of the SEQUENCE propagator increases with larger widths. For width 128, the SEQUENCE propagator can achieve orders of magnitude smaller search trees and solving time than the AMONG propagators, which again demonstrates the advantage of MDD propagation for SEQUENCE when compared with the AMONG decomposition.

Chapter 11
Sequencing and Single-Machine Scheduling

Abstract In this chapter we provide an in-depth study of representing and handling single-machine scheduling and sequencing problems with decision diagrams. We provide exact and relaxed MDD representations, together with MDD filtering algorithms for various side constraints, including time windows, precedence constraints, and sequence-dependent setup times. We extend a constraint-based scheduling solver with these techniques, and provide an experimental evaluation for a wide range of problems, including the traveling salesman problem with time windows, the sequential ordering problem, and minimum-tardiness sequencing problems. The results demonstrate that MDD propagation can improve a state-of-the-art constraint-based scheduler by orders of magnitude in terms of solving time.

11.1 Introduction

Sequencing problems are among the most widely studied problems in operations research. Specific variations of sequencing problems include single-machine scheduling, the traveling salesman problem with time windows, and precedence-constrained machine scheduling. Sequencing problems are those where the best order for performing a set of tasks must be determined, which in many cases leads to an NP-hard problem [71, Section A5]. Sequencing problems are prevalent in manufacturing and routing applications, including production plants where jobs should be processed one at a time on an assembly line, and in mail services where packages must be scheduled for delivery on a vehicle. Industrial problems that involve multiple facilities may also be viewed as sequencing problems in certain scenarios, e.g., when

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9_11

a machine is the bottleneck of a manufacturing plant [128]. Existing methods for sequencing problems either follow a dedicated heuristic for a specific problem class or utilize a generic solving methodology such as integer programming or constraint programming.

In this chapter we present a new approach for solving sequencing problems, based on MDDs. We argue that relaxed MDDs can be particularly useful as a discrete relaxation of the feasible set of sequencing problems. We focus on a broad class of sequencing problems where jobs should be scheduled on a single machine and are subject to precedence and time window constraints, and in which setup times can be present. It generalizes a number of single-machine scheduling problems and variations of the traveling salesman problem (TSP). The relaxation provided by the MDD, however, is suitable for any problem where the solution is defined by a permutation of a fixed number of tasks, and it does not directly depend on particular constraints or on the objective function.

The structure of this chapter is as follows: We first introduce a representation of the feasible set of a sequencing problem as an MDD, and show how we can obtain a relaxed MDD. We then show how the relaxed MDD can be used to compute bounds on typical objective functions in scheduling, such as the makespan and total tardiness. Moreover, we describe how to derive more structured sequencing information from the relaxed MDD, in particular a valid set of precedence relations that must hold in any feasible solution.

We also propose a number of techniques for strengthening the MDD relaxation, which take into account the precedence and time window constraints. We demonstrate that these generic techniques can be used to derive a polynomial-time algorithm for a particular TSP variant introduced by [14] by showing that the associated MDD has polynomial size.

To demonstrate the use of relaxed MDDs in practice, we apply our techniques to *constraint-based scheduling* [17]. Constraint-based scheduling plays a central role as a general-purpose methodology in complex and large-scale scheduling problems. Examples of commercial applications that apply this methodology include yard planning of the Singapore port and gate allocation of the Hong Kong airport [68], Brazilian oil-pipeline scheduling [112], and home healthcare scheduling [136]. We show that, by using the relaxed MDD techniques described here, we can improve the performance of the state-of-the-art constraint-based schedulers by orders of magnitude on single-machine problems without losing the generality of the method.

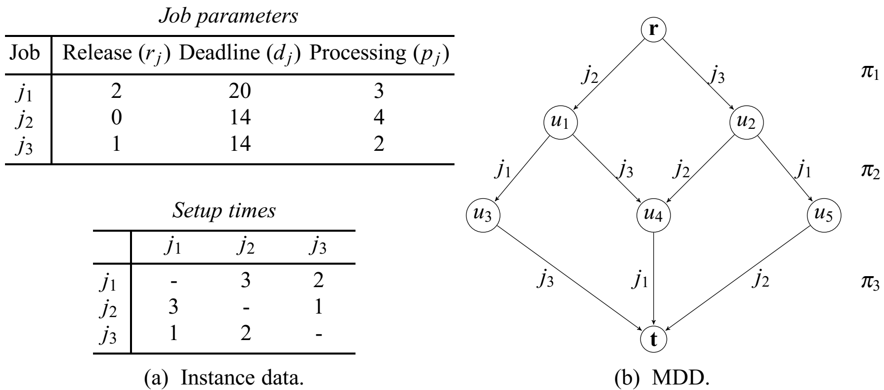
11.2 Problem Definition

As mentioned above, we focus on generic sequencing problems, presented here in terms of 'unary machine' scheduling. Note that a machine may refer to any resource capable of handling at most one activity at a time.

Let $\mathscr{J} = \{j_1, \ldots, j_n\}$ be a set of n jobs to be processed on a machine that can perform at most one job at a time. Each job $j \in \mathscr{J}$ has an associated *processing time* p_j, which is the number of time units the job requires from the machine, and a *release date* r_j, the time from which job j is available to be processed. For each pair of distinct jobs $j, j' \in \mathscr{J}$ a *setup time* $t_{j,j'}$ is defined, which indicates the minimum time that must elapse between the end of j and the beginning of j' if j' is the first job processed after j finishes. We assume that jobs are *non-preemptive*, i.e., we cannot interrupt a job while it is being processed on the machine.

We are interested in assigning a *start time* $s_j \geq r_j$ for each job $j \in \mathscr{J}$ such that job processing intervals do not overlap, the resulting schedule observes a number of constraints, and an objective function f is minimized. Two types of constraints are considered in this chapter: *precedence constraints*, requiring that $s_j \leq s_{j'}$ for certain pairs of jobs $(j, j') \in \mathscr{J} \times \mathscr{J}$, which we equivalently write $j \ll j'$; and *time window constraints*, where the *completion time* $c_j = s_j + p_j$ of each job $j \in \mathscr{J}$ must be such that $c_j \leq d_j$ for some *deadline* d_j. Furthermore, we study three representative objective functions in scheduling: the *makespan*, where we minimize the completion time of the schedule, or $\max_{j \in \mathscr{J}} c_j$; the *total tardiness*, where we minimize $\sum_{j \in \mathscr{J}} (\max\{0,\ c_j - \delta_j\})$ for given *due dates* δ_j; and the *sum of setup times*, where we minimize the value obtained by accumulating the setup times $t_{j,j'}$ for all consecutive jobs j, j' in a schedule. Note that for these objective functions we can assume that jobs should always be processed as early as possible (i.e., idle times do not decrease the value of the objective function).

Since jobs are processed one at a time, any solution to such scheduling problem can be equivalently represented by a total ordering $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of $\mathscr{J}$. The start time of job j implied by π is given by $s_j = r_j$ if $j = \pi_1$, and $s_j = \max\{r_j,\ s_{\pi_{i-1}} + p_{\pi_{i-1}} + t_{\pi_{i-1},j}\}$ if $j = \pi_i$ for some $i \in \{2, \ldots, n\}$. We say that an ordering π of $\mathscr{J}$ is *feasible* if the implied job times observe the precedence and time window constraints, and *optimal* if it is feasible and minimizes f.



Job parameters

Job	Release (r_j)	Deadline (d_j)	Processing (p_j)
j_1	2	20	3
j_2	0	14	4
j_3	1	14	2

Setup times

	j_1	j_2	j_3
j_1	-	3	2
j_2	3	-	1
j_3	1	2	-

(a) Instance data. (b) MDD.

Fig. 11.1 Example of an MDD for a sequencing problem.

11.3 MDD Representation

We have already seen examples of single-machine scheduling in Section 3.8 and Section 8.3, and we will follow a similar MDD representation in this chapter. That is, we define an MDD $\mathcal{M}$ whose paths represent the feasible orderings of $\mathcal{J}$. The set of nodes of $\mathcal{M}$ are partitioned into $n+1$ layers $L_1,\dots,L_{n+1}$, where layer L_i corresponds to the i-th position π_i of the feasible orderings encoded by $\mathcal{M}$, for $i=1,\dots,n$. Layers L_1 and L_{n+1} are singletons representing the root $\mathbf{r}$ and the terminal $\mathbf{t}$, respectively. In this chapter, an arc $a=(u,v)$ of $\mathcal{M}$ is always directed from a source node u in some layer L_i to a target node v in the *subsequent* layer L_{i+1}, $i\in\{1,\dots,n\}$. We write $\ell(a)$ to indicate the layer of the source node u of the arc a (i.e., $u\in L_{\ell(a)}$).

With each arc a of $\mathcal{M}$ we associate a label $d(a)\in\mathcal{J}$ that represents the assignment of the job $d(a)$ to the $\ell(a)$-th position of the orderings identified by the paths traversing a. Hence, an arc-specified path $(a_1,\dots,a_n)$ from $\mathbf{r}$ to $\mathbf{t}$ identifies the ordering $\pi=(\pi_1,\dots,\pi_n)$, where $\pi_i=d(a_i)$ for $i=1,\dots,n$. Every feasible ordering is identified by some path from $\mathbf{r}$ to $\mathbf{t}$ in $\mathcal{M}$, and conversely every path from $\mathbf{r}$ to $\mathbf{t}$ identifies a feasible ordering.

Example 11.1. We provide an MDD representation for a sequencing problem with three jobs j_1, j_2, and j_3. The instance data are presented in Fig. 11.1(a), and the associated MDD $\mathcal{M}$ is depicted in Fig. 11.1(b). No precedence constraints are considered. There are four feasible orderings in total, each identified by a path from $\mathbf{r}$ to $\mathbf{t}$ in $\mathcal{M}$. In particular, the path traversing nodes $\mathbf{r}$, u_2, u_4, and $\mathbf{t}$ represents a

solution where jobs j_3, j_2, and j_1 are performed in this order. The completion times for this solution are $c_{j_1} = 15$, $c_{j_2} = 9$, and $c_{j_3} = 3$. Note that we can never have a solution where j_1 is first on the machine, otherwise either the deadline of j_2 or j_3 would be violated. Hence, there is no arc a with $d(a) = j_1$ directed out of $\mathbf{r}$.

We next show how to compute the orderings that yield the optimal makespan and the optimal sum of setup times in polynomial time in the size of $\mathscr{M}$. For the case of total tardiness and other similar objective functions, we are able to provide a lower bound on its optimal value also in polynomial time in $\mathscr{M}$.

- *Makespan.* For each arc a in $\mathscr{M}$, define the *earliest completion time* of a, or ect_a, as the minimum completion time of the job $d(a)$ among all orderings that are identified by the paths in $\mathscr{M}$ containing a. If the arc a is directed out of $\mathbf{r}$, then a assigns the first job that is processed in such orderings, thus $ect_a = r_{d(a)} + p_{d(a)}$. For the remaining arcs, recall that the completion time c_{π_i} of a job π_i depends only on the completion time of the previous job π_{i-1}, the setup time t_{π_{i-1},π_i}, and on the specific job parameters; namely, $c_{\pi_i} = \max\{r_{\pi_i},\ c_{\pi_{i-1}} + t_{\pi_{i-1},\pi_i}\} + p_{\pi_i}$. It follows that the earliest completion time of an arc $a = (u,v)$ can be computed by the relation

$$ect_a = \max\{r_{d(a)},\ \min\{ect_{a'} + t_{d(a'),d(a)} : a' \in in(u)\}\} + p_{d(a)}. \tag{11.1}$$

  The minimum makespan is given by $\min_{a \in in(\mathbf{t})} ect_a$, as the arcs directed to $\mathbf{t}$ assign the last job in all orderings represented by $\mathscr{M}$. An optimal ordering can be obtained by recursively retrieving the minimizer arc $a' \in in(u)$ in the “min” of (11.1).
- *Sum of Setup Times.* The minimum sum of setup times is computed analogously: For an arc $a = (u,v)$, let st_a represent the minimum sum of setup times up to job $d(a)$ among all orderings that are represented by the paths in $\mathscr{M}$ containing a. If a is directed out of $\mathbf{r}$, we have $st_a = 0$; otherwise,

$$st_a = \min\{st_{a'} + t_{d(a'),d(a)} : a' \in in(u)\}. \tag{11.2}$$

  The minimum sum of setup times is given by $\min_{a \in in(\mathbf{t})} st_a$.
- *Total Tardiness.* The *tardiness* of a job j is defined by $\max\{0,\ c_j - \delta_j\}$ for some due date δ_j. Unlike the previous two cases, the tardiness value that a job attains in an optimal solution depends on the sequence of all activities, not only on its individual contribution or the value of its immediate predecessor. Nonetheless, as

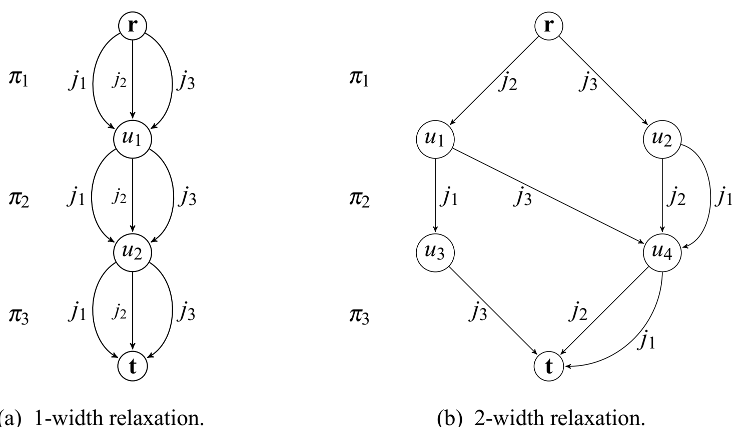
the tardiness function for a job is nondecreasing in its completion time, we can utilize the earliest completion time as follows: For any arc $a = (u, v)$, the value $\max\{0,\ ect_a - \delta_{d(a)}\}$ yields a lower bound on the tardiness of the job $d(a)$ among all orderings that are represented by the paths in $\mathscr{M}$ containing a. Hence, a lower bound on the total tardiness is given by the length of the shortest path from $\mathbf{r}$ to $\mathbf{t}$, where the length of an arc a is set to $\max\{0,\ ect_a - \delta_{d(a)}\}$. Observe that this bound is tight if the MDD is composed by a single path.

We remark that valid bounds for many other types of objective in the scheduling literature can be computed in an analogous way as above. For example, suppose the objective is to minimize $\sum_{j \in \mathscr{J}} f_j(c_j)$, where f_j is a function defined for each job j and which is nondecreasing in the completion time c_j. Then, as in total tardiness, the value $f_{d(a)}(ect_a)$ for an arc $a = (u, v)$ yields a lower bound on the minimum value of $f_{d(a)}(c_{d(a)})$ among all orderings that are identified by the paths in $\mathscr{M}$ containing a. Using such bounds as arc lengths, the shortest path from $\mathbf{r}$ to $\mathbf{t}$ represents a lower bound on $\sum_{j \in \mathscr{J}} f_j(c_j)$. This bound is tight if $f_j(c_j) = c_j$, or if $\mathscr{M}$ is composed by a single path. Examples of such objectives include weighted total tardiness, total square tardiness, sum of (weighted) completion times, and number of late jobs.

Example 11.2. In the instance depicted in Fig. 11.1, we can apply the recurrence relation (11.1) to obtain $ect_{\mathbf{r},u_1} = 4$, $ect_{\mathbf{r},u_2} = 3$, $ect_{u_1,u_3} = 10$, $ect_{u_1,u_4} = 7$, $ect_{u_2,u_4} = 9$, $ect_{u_2,u_5} = 7$, $ect_{u_3,\mathbf{t}} = 14$, $ect_{u_4,\mathbf{t}} = 11$, and $ect_{u_5,\mathbf{t}} = 14$. The optimal makespan is $\min\{ect_{u_3,\mathbf{t}}, ect_{u_4,\mathbf{t}}, ect_{u_5,\mathbf{t}}\} = ect_{u_4,\mathbf{t}} = 11$; it corresponds to the path $(\mathbf{r}, u_1, u_4, \mathbf{t})$, which identifies the optimal ordering (j_2, j_3, j_1). The same ordering also yields the optimal sum of setup times with a value of 2.

Suppose now that we are given due dates $\delta_{j_1} = 13$, $\delta_{j_2} = 8$, and $\delta_{j_3} = 3$. The length of an arc a is given by $l_a = \max\{0,\ ect_a - \delta_{d(a)}\}$, as described earlier. We have $l_{u_1,u_4} = 4$, $l_{u_2,u_4} = 1$, $l_{u_3,\mathbf{t}} = 11$, and $l_{u_5,\mathbf{t}} = 6$; all remaining arcs a are such that $l_a = 0$. The shortest path in this case is $(\mathbf{r}, u_2, u_4, \mathbf{t})$ and has a value of 1. The minimum tardiness, even though it is given by the ordering identified by this same path, (j_3, j_2, j_1), has a value of 3.

The reason for this gap is that the ordering with minimum tardiness does not necessarily coincide with the schedule corresponding to the earliest completion time. Namely, we computed $l_{u_4,\mathbf{t}} = 0$ considering $ect_{u_4,\mathbf{t}} = 11$, since the completion time of the job $d(u_4, \mathbf{t}) = j_1$ is 11 in (j_2, j_3, j_1). However, in the optimal ordering (j_3, j_2, j_1) for total tardiness, the completion time of j_1 would be 15; this solution yields a better cost than (j_2, j_3, j_1) due to the reduction in the tardiness of j_3.



(a) 1-width relaxation. (b) 2-width relaxation.

Fig. 11.2 Two relaxed MDDs for the sequencing problem in Fig. 11.1.

11.4 Relaxed MDDs

We next consider the compilation of relaxed MDDs for sequencing problems, which represent a superset of the feasible orderings of $\mathscr{J}$. As an illustration, Fig. 11.2(a) and 11.2(b) present two examples of a relaxed MDD with maximum width $W = 1$ and $W = 2$, respectively, for the problem depicted in Fig. 11.1. In particular, the MDD in Fig. 11.2(a) encodes all the orderings represented by permutations of $\mathscr{J}$ with repetition, hence it trivially contains the feasible orderings of any sequencing problem. It can be generally constructed as follows: We create one node u_i for each layer L_i and connect the pair of nodes u_i and u_{i+1}, $i = 1, \dots, n$, with arcs $a_1, \dots, a_n$ such that $d(a_l) = j_k$ for each job j_k.

It can also be verified that the MDD in Fig. 11.2(b) contains all the feasible orderings of the instance in Fig. 11.1. However, the rightmost path going through nodes **r**, u_2, u_4, and **t** identifies an ordering $\pi = (j_3, j_1, j_1)$, which is infeasible as job j_1 is assigned twice in π.

The procedures in Section 11.3 for computing the optimal makespan and the optimal sum of setup times now yield a lower bound on such values when applied to a relaxed MDD, since all feasible orderings of $\mathscr{J}$ are encoded in the diagram. Moreover, the lower bounding technique for total tardiness remains valid.

Considering that a relaxed MDD $\mathscr{M}$ can be easily constructed for any sequencing problem (e.g., the 1-width relaxation of Fig. 11.2(a)), we can now apply the techniques presented in Section 4.7 and Chapter 9 to incrementally modify $\mathscr{M}$ in order to strengthen the relaxation it provides, while observing the maximum width

W. Under certain conditions, we obtain the reduced MDD representing exactly the feasible orderings of $\mathscr{J}$, provided that W is sufficiently large.

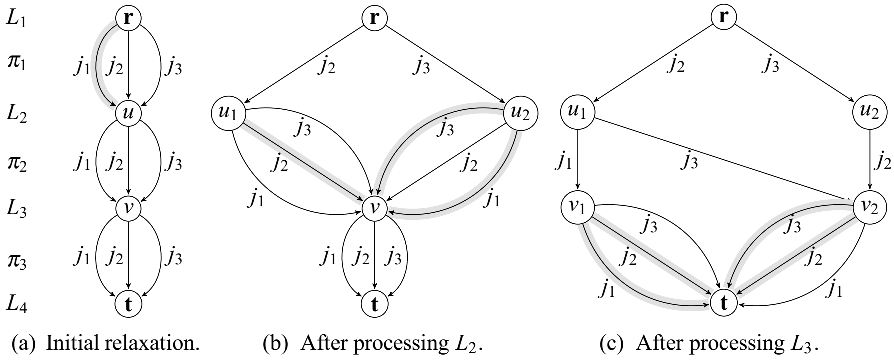
Recall that we modify a relaxed MDD $\mathscr{M}$ by applying the operations of *filtering* and *refinement*, which aim at approximating $\mathscr{M}$ to an *exact* MDD, i.e., one that exactly represents the feasible orderings of $\mathscr{J}$. We revisit these concepts below, and describe them in the context of sequencing problems:

- *Filtering*. An arc a in $\mathscr{M}$ is *infeasible* if all the paths in $\mathscr{M}$ containing a represent orderings that are not feasible. Filtering consists of identifying infeasible arcs and removing them from $\mathscr{M}$, which would hence eliminate one or more infeasible orderings that are encoded in $\mathscr{M}$. We will provide details on the filtering operation in Section 11.5.
- *Refinement*. A relaxed MDD can be intuitively perceived as a diagram obtained by merging nonequivalent nodes of an exact MDD for the problem. Refinement consists of identifying these nodes in $\mathscr{M}$ that are encompassing multiple equivalence classes, and *splitting* them into two or more new nodes to represent such classes more accurately (as long as the maximum width W is not violated). In particular, a node u in layer L_i can be split if there exist two partial orderings π_1', π_2' identified by paths from $\mathbf{r}$ to u such that, for some $\pi^* = (\pi_i, \ldots, \pi_n)$, (π_1', π^*) is a feasible ordering while (π_2', π^*) is not. If this is the case, then the partial paths in $\mathscr{M}$ representing such orderings must end in different nodes of the MDD, which will be necessarily nonequivalent by definition. We will provide details on the refinement operation in Section 11.7.

Observe that, if a relaxed MDD $\mathscr{M}$ does not have any infeasible arcs and no nodes require splitting, then by definition $\mathscr{M}$ is exact. However, it may not necessarily be reduced.

As mentioned in Chapter 9, filtering and refinement are independent operations that can be applied to $\mathscr{M}$ in any order that is suitable for the problem at hand. In this chapter we assume a top-down approach: We traverse layers $L_2, \ldots, L_{n+1}$ one at a time in this order. At each layer L_i, we first apply filtering to remove infeasible arcs that are directed to the nodes in L_i. After the filtering is complete, we perform refinement to split the nodes in layer L_i as necessary, while observing the maximum width W.

Example 11.3. Figure 11.3 illustrates the top-down application of filtering and refinement for layers L_2 and L_3. Assume a scheduling problem with three jobs $\mathscr{J} = \{j_1, j_2, j_3\}$ and subject to a single precedence constraint stating that job j_2



(a) Initial relaxation. (b) After processing L_2. (c) After processing L_3.

Fig. 11.3 Example of filtering and refinement. The scheduling problem is such that job j_2 must precede j_1 in all feasible orderings. Shaded arrows represent infeasible arcs detected by the filtering.

must precede job j_1. The initial relaxed MDD is a 1-width relaxation, depicted in Fig. 11.3(a). Our maximum width is set to $W = 2$.

We start by processing the incoming arcs at layer L_2. The filtering operation detects that the arc $a \in in(u)$ with $d(a) = j_1$ is infeasible, otherwise we will have an ordering starting with job j_1, violating the precedence relation. Refinement will split node u into nodes u_1 and u_2, since for any feasible ordering starting with job j_2, i.e., (j_2, π') for some π', the ordering (j_3, π') is infeasible as it will necessarily assign job j_3 twice. The resulting MDD is depicted in Fig. 11.3(b). Note that, when a node is split, we replicate its outgoing arcs to each of the new nodes.

We now process the incoming arcs at layer L_3. The filtering operation detects that the arc with label j_2 directed out of u_1 and the arc with label j_3 directed out of u_2 are infeasible, since the corresponding paths from **r** to v would yield orderings that assign some job twice. The arc with label j_1 leaving node u_2 is also infeasible, since we cannot have any ordering with prefix (j_3, j_1). Finally, refinement will split node v into nodes v_1 and v_2; note in particular that the feasible orderings prefixed by (j_2, j_3) and (j_3, j_2) have the same completions, namely (j_1), therefore the corresponding paths end at the same node v_1. The resulting MDD is depicted in Fig. 11.3(c). We can next process the incoming arcs at layer L_4, and remove arcs with labels j_1 and j_2 out of v_1, and arcs with labels j_2 and j_3 out of v_2.

11.5 Filtering

We next apply the methodology developed in Chapter 9 to sequencing problems. That is, for each constraint type of our problem, we identify necessary conditions for the infeasibility of an arc in $\mathscr{M}$. To this end, for each constraint type $\mathscr{C}$, we equip the nodes and arcs of $\mathscr{M}$ with state information that is specific to $\mathscr{C}$.

We take care that the conditions for infeasibility can be tested in polynomial time in the size of the relaxed MDD $\mathscr{M}$. Namely, we restrict our state definitions to have size $O(|\mathscr{J}|)$ and to be Markovian, in that they only depend on the states of the nodes and arcs in the adjacent layers. Thus, the states can be computed simultaneously with the filtering and refinement operations during the top-down approach described in Section 11.4. We also utilize additional state information that is obtained through an extra bottom-up traversal of the MDD and that, when combined with the top-down states, leads to stronger tests.

11.5.1 Filtering Invalid Permutations

The feasible orderings of any sequencing problem are permutations of $\mathscr{J}$ without repetition, which can be enforced with the constraint ALLDIFFERENT$(\pi_1,\ldots,\pi_n)$. Hence, we can directly use the filtering conditions for this constraint described in Section 4.7.1, based on the states $All_u^{\downarrow} \subseteq \mathscr{J}$ and $Some_u^{\downarrow} \subseteq \mathscr{J}$ for each node u of $\mathscr{M}$. Recall that the state $All_u^{\downarrow}$ is the set of arc labels that appear in *all* paths from the root node $\mathbf{r}$ to u, while the state $Some_u^{\downarrow}$ is the set of arc labels that appear in *some* path from the root node $\mathbf{r}$ to u. As presented in Section 4.7.1, an arc $a = (u,v)$ is infeasible if either $d(a) \in All_u^{\downarrow}$ (condition 4.6) or $|Some_u^{\downarrow}| = \ell(a)$ and $d(a) \in Some_u^{\downarrow}$ (condition 4.7).

We also equip the nodes with additional states that can be derived from a bottom-up perspective of the MDD. Namely, we define two new states $All_u^{\uparrow} \subseteq \mathscr{J}$ and $Some_u^{\uparrow} \subseteq \mathscr{J}$ for each node u of $\mathscr{M}$. They are similar to the states $All_u^{\downarrow}$ and $Some_u^{\downarrow}$, but now they are computed with respect to the paths from $\mathbf{t}$ to u instead of the paths from $\mathbf{r}$ to u, and analogously computed recursively.

It follows from Section 4.7.1 that an arc $a = (u,v)$ is infeasible if we have either $d(a) \in All_v^{\uparrow}$ (condition 4.10), $|Some_v^{\uparrow}| = n - \ell(a)$ and $d(a) \in Some_v^{\uparrow}$ (condition 4.11), or $|Some_u^{\downarrow} \cup \{d(a)\} \cup Some_v^{\uparrow}| < n$ (condition 4.12).

11.5.2 Filtering Precedence Constraints

We next consider filtering a given set of precedence constraints, where we write $j \ll j'$ if a job j should precede job j' in any feasible ordering. We assume the precedence relations are not trivially infeasible, i.e., there are no cycles of the form $j \ll j_1 \ll \cdots \ll j_m \ll j$. We can apply the same states defined for the ALLDIFFERENT constraint in Section 11.5.1 for this particular case.

Lemma 11.1. *An arc $a = (u, v)$ with label $d(a)$ is infeasible if either of the following conditions hold:*

$$\exists\, j \in (\mathscr{J} \setminus Some^{\downarrow}_{u}) \text{ s.t. } j \ll d(a), \tag{11.3}$$

$$\exists\, j \in (\mathscr{J} \setminus Some^{\uparrow}_{v}) \text{ s.t. } d(a) \ll j. \tag{11.4}$$

Proof. Let π' be any partial ordering identified by a path from $\mathbf{r}$ to u, and consider (11.3). By definition of $Some^{\downarrow}_{u}$, we have that any job j in the set $\mathscr{J} \setminus Some^{\downarrow}_{u}$ is not assigned to any position in π'. Thus, if any such job j must precede $d(a)$, then all orderings prefixed by $(\pi', d(a))$ will violate this precedence constraint, and the arc is infeasible. The condition (11.4) is the symmetrical version of (11.3). □

11.5.3 Filtering Time Window Constraints

Consider now that a deadline d_j is imposed for each job $j \in \mathscr{J}$. With each arc a we associate the state ect_a as defined in Section 11.3: It corresponds to the minimum completion time of the job in the $\ell(a)$-th position among all orderings that are identified by paths in $\mathscr{M}$ containing the arc a. As in relation (11.1), the state ect_a for an arc $a = (u, v)$ is given by the recurrence

$$ect_a = \begin{cases} r_{d(a)} + p_{d(a)} & \text{if } a \in out(\mathbf{r}), \\ \max\{r_{d(a)}, \min\{ect_{a'} + t_{d(a'),d(a)} : a' \in in(u), d(a) \neq d(a')\}\} + p_{d(a)} & \\ & \text{otherwise.} \end{cases}$$

Here we added the trivial condition $d(a) \neq d(a')$ to strengthen the bound on ect_a in the relaxed MDD $\mathscr{M}$. We could also include the condition $d(a) \not\ll d(a')$ if precedence constraints are imposed over $d(a)$.

We next consider a symmetrical version of ect_a to derive a necessary infeasibility condition for time window constraints. Namely, with each arc a we associate the state lst_a, which represents the *latest start time* of a: For all orderings that are identified by paths in $\mathscr{M}$ containing the arc a, the value lst_a corresponds to an upper bound on the maximum start time of the job in the $\ell(a)$-th position so that no deadlines are violated in such orderings. The state lst_a for an arc $a = (u,v)$ is given by the following recurrence, which can be computed through a single bottom-up traversal of $\mathscr{M}$:

$$lst_a = \begin{cases} d_{d(a)} - p_{d(a)} & \text{if } a \in in(\mathbf{t}), \\ \min\{d_{d(a)},\ \max\{lst_{a'} - t_{d(a),d(a')} : a' \in out(v), d(a) \neq d(a')\}\} - p_{d(a)} & \\ & \text{otherwise.} \end{cases}$$

We combine ect_a and lst_a to derive the following rule:

Lemma 11.2. *An arc $a = (u,v)$ is infeasible if*

$$ect_a > lst_a + p_{d(a)}. \tag{11.5}$$

Proof. The value $lst_a + p_{d(a)}$ represents an upper bound on the maximum time the job $d(a)$ can be completed so that no deadlines are violated in the orderings identified by paths in $\mathscr{M}$ containing a. Since ect_a is the minimum time that job $d(a)$ will be completed among all such orderings, no feasible ordering identified by a path traversing a exists if rule (11.5) holds. □

11.5.4 Filtering Objective Function Bounds

As described in Section 9.2, in constraint programming systems the objective function is treated as a constraint $z \leq z^*$, where z represents the objective function, and z^* is an upper bound of the objective function value. The upper bound typically corresponds to the best feasible solution found during the search for an optimal solution.

Below we describe filtering procedures for the 'objective constraint' for sequencing problems. Given z^*, an arc a is infeasible with respect to the objective if all paths in $\mathscr{M}$ that contain a have objective value greater than z^*. However, the associated filtering method depends on the form of the objective function. We consider here

three types of objectives: minimize makespan, minimize the sum of setup times, and minimize total (weighted) tardiness.

Minimize Makespan

If the objective is to minimize makespan, we can replace the deadline d_j by $d'_j = \min\{d_j, z^*\}$ for all jobs j and apply the same infeasibility condition as in Lemma 11.2.

Minimize Sum of Setup Times

If z^* represents an upper bound on the sum of setup times, we proceed as follows: For each arc $a = (u,v)$ in $\mathscr{M}$, let $st_a^{\downarrow}$ be the minimum possible sum of setup times incurred by the partial orderings represented by paths from $\mathbf{r}$ to v that contain a. We recursively compute

$$st_a^{\downarrow} = \begin{cases} 0, & \text{if } a \in out(\mathbf{r}), \\ \min\{t_{d(a'),d(a)} + st_{a'}^{\downarrow} : a' \in in(u), d(a) \neq d(a')\}, & \text{otherwise.} \end{cases}$$

Now, for each arc $a = (u,v)$ let $st_a^{\uparrow}$ be the minimum possible sum of setup times incurred by the partial orderings represented by paths from u to $\mathbf{t}$ that contain a. The state $st_a^{\uparrow}$ can be recursively computed through a bottom-up traversal of $\mathscr{M}$, as follows:

$$st_a^{\uparrow} = \begin{cases} 0, & \text{if } a \in in(\mathbf{t}), \\ \min\{t_{d(a),d(a')} + st_{a'}^{\uparrow} : a' \in out(v), d(a) \neq d(a')\}, & \text{otherwise.} \end{cases}$$

Lemma 11.3. *An arc a is infeasible if*

$$st_a^{\downarrow} + st_a^{\uparrow} > z^*. \tag{11.6}$$

Proof. It follows directly from the definitions of $st_a^{\downarrow}$ and $st_a^{\uparrow}$. □

Minimize Total Tardiness

To impose an upper bound z^* on the total tardiness, assume ect_a is computed for each arc a. We define the length of an arc a as $l_a = \max\{0, ect_a - \delta_{d(a)}\}$. For a node u, let $sp_u^{\downarrow}$ and $sp_u^{\uparrow}$ be the shortest path from $\mathbf{r}$ to u and from $\mathbf{t}$ to u, respectively, with respect to the lengths l_a. That is,

$$sp_u^{\downarrow} = \begin{cases} 0, & \text{if } u = \mathbf{r}, \\ \min\{l_a + sp_v^{\downarrow} : a = (v,u) \in in(u)\}, & \text{otherwise} \end{cases}$$

and

$$sp_u^{\uparrow} = \begin{cases} 0, & \text{if } u = \mathbf{t}, \\ \min\{l_a + sp_v^{\uparrow} : a = (u,v) \in out(u)\}, & \text{otherwise.} \end{cases}$$

Lemma 11.4. *A node u should be removed from $\mathscr{M}$ if*

$$sp_u^{\downarrow} + sp_u^{\uparrow} > z^*. \tag{11.7}$$

Proof. Length l_a represents a lower bound on the tardiness of job $d(a)$ with respect to solutions identified by $\mathbf{r}$–$\mathbf{t}$ paths that contain a. Thus, $sp_u^{\downarrow}$ and $sp_u^{\uparrow}$ are a lower bound on the total tardiness for the partial orderings identified by paths from $\mathbf{r}$ to u and $\mathbf{t}$ to u, respectively, since the tardiness of a job is nondecreasing in its completion time. □

11.6 Inferring Precedence Relations from Relaxed MDDs

Given a set of precedence relations for a problem (e.g., that were possibly derived from other relaxations), we can use the filtering rules (11.3) and (11.4) from Section 11.5.2 to strengthen a relaxed MDD. In this section, we show that a converse relation is also possible. Namely, given a relaxed MDD $\mathscr{M}$, we can deduce all precedence relations that are satisfied by the partial orderings represented by $\mathscr{M}$ in polynomial time in the size of $\mathscr{M}$. To this end, assume that the states $All_u^{\downarrow}$, $All_u^{\uparrow}$, $Some_u^{\downarrow}$, and $Some_u^{\uparrow}$ as described in Section 11.5.1 are computed for all nodes u in $\mathscr{M}$. We have the following results:

Theorem 11.1. *Let $\mathscr{M}$ be an MDD that exactly identifies all the feasible orderings of $\mathscr{J}$. A job j must precede job j' in any feasible ordering if and only if either $j' \notin All_u^{\downarrow}$ or $j \notin All_u^{\uparrow}$ for all nodes u in $\mathscr{M}$.*

Proof. Suppose there exists a node u in layer L_i, $i \in \{1,\dots,n+1\}$, such that $j' \in All_u^{\downarrow}$ and $j \in All_u^{\uparrow}$. By definition, there exists a path $(\mathbf{r},\dots,u,\dots,\mathbf{t})$ that identifies an ordering where job j' starts before job j. This is true if and only if job j does not precede j' in any feasible ordering. □

Corollary 11.1. *The set of all precedence relations that must hold in any feasible ordering can be extracted from $\mathscr{M}$ in $O(n^2\,|\mathscr{M}|)$.*

Proof. Construct a digraph $G^* = (\mathscr{J}, E^*)$ by adding an arc (j, j') to E^* if and only if there exists a node u in $\mathscr{M}$ such that $j' \in All_u^{\downarrow}$ and $j \in All_u^{\uparrow}$. Checking this condition for all pairs of jobs takes $O(n^2)$ for each node in $\mathscr{M}$, and hence the time complexity to construct G^* is $O(n^2|\mathscr{M}|)$. According to Theorem 11.1 and the definition of G^*, the complement graph of G^* contains an edge (j, j') if and only if $j \ll j'$. □

As we are mainly interested in relaxed MDDs, we derive an additional corollary of Theorem 11.1.

Corollary 11.2. *Given a relaxed MDD $\mathscr{M}$, an activity j must precede activity j' in any feasible solution if $(j' \notin Some_u^{\downarrow})$ or $(j \notin Some_u^{\uparrow})$ for all nodes u in $\mathscr{M}$.*

Proof. It follows from the state definitions that $All_u^{\downarrow} \subseteq Some_u^{\downarrow}$ and $All_u^{\uparrow} \subseteq Some_u^{\uparrow}$. Hence, if the conditions for the relation $j \ll j'$ from Theorem 11.1 are satisfied by $Some_u^{\downarrow}$ and $Some_v^{\uparrow}$, they must be also satisfied by any MDD which only identifies feasible orderings. □

By Corollary 11.2, the precedence relations implied by the solutions of a relaxed MDD $\mathscr{M}$ can be extracted by applying the algorithm in Corollary 11.1 to the states $Some_v^{\downarrow}$ and $Some_v^{\uparrow}$. Since $\mathscr{M}$ has at most $O(nW)$ nodes and $O(nW^2)$ arcs, the time to extract the precedences has a worst-case complexity of $O(n^3W^2)$ by the presented algorithm. These precedences can then be used for guiding search or communicated to other methods or relaxations that may benefit from them.

11.7 Refinement

As in Section 4.7.1.2, we will develop a refinement procedure based on the permutation structure of the jobs, represented by the ALLDIFFERENT constraint. The goal of our heuristic refinement is to be as precise as possible with respect to the equivalence classes that refer to jobs with a higher *priority*, where the priority of a

job follows from the problem data. More specifically, we will develop a heuristic for refinement that, when combined with the infeasibility conditions for the permutation structure described in Section 11.5.1, yields a relaxed MDD where the jobs with a high priority are represented exactly with respect to that structure; that is, these jobs are assigned to exactly one position in all orderings encoded by the relaxed MDD. We also take care that a given maximum width W is observed when creating new nodes in a layer.

Thus, if higher priority is given to jobs that play a greater role in the feasibility or optimality of the sequencing problem at hand, the relaxed MDD may represent more accurately the feasible orderings of the problem, providing, e.g., better bounds on the objective function value. For example, suppose we wish to minimize the makespan on an instance where certain jobs have very large release dates and processing times in comparison with other jobs. If we construct a relaxed MDD where these longer jobs are assigned exactly once in all orderings encoded by the MDD, the bound on the makespan would be potentially tighter with respect to the ones obtained from other possible relaxed MDDs for this same instance. Examples of job priorities for other objective functions are presented in Section 11.9. Recall from Section 4.7.1.2 that the refinement heuristic requires a *ranking* of jobs $\mathscr{J}^* = \{j^*_1, \dots, j^*_n\}$, where jobs with smaller index in $\mathscr{J}^*$ have higher priority.

We note that the refinement heuristic also yields a *reduced* MDD $\mathscr{M}$ for certain structured problems, given a sufficiently large width. The following corollary, stated without proof, is directly derived from Lemma 4.4 and Theorem 4.3.

Corollary 11.3. *Assume $W = +\infty$. For a sequencing problem having only precedence constraints, the relaxed MDD $\mathscr{M}$ that results from the constructive proof of Theorem 4.3 is a reduced MDD that exactly represents the feasible orderings of this problem.*

Lastly, recall that equivalence classes corresponding to constraints other than the permutation structure may also be taken into account during refinement. Therefore, if the maximum width W is not met in the refinement procedure above, we assume that we will further split nodes by arbitrarily partitioning their incoming arcs. Even though this may yield false equivalence classes, the resulting $\mathscr{M}$ is still a valid relaxation and may provide a stronger representation.

11.8 Encoding Size for Structured Precedence Relations

The actual constraints that define a problem instance greatly impact the size of an MDD. If these constraints carry a particular structure, we may be able to compactly represent that structure in an MDD, perhaps enabling us to bound its width.

In this section we present one such case for a problem class introduced by [14], in which jobs are subject to *discrepancy* precedence constraints: For a fixed parameter $k \in \{1,\ldots,n\}$, the relation $j_p \ll j_q$ must be satisfied for any two jobs $j_p, j_q \in \mathscr{J}$ if $q \geq p+k$. This precedence structure was motivated by a real-world application in steel rolling mill scheduling. The work by [15] also demonstrates how solution methods to this class of problems can serve as auxiliary techniques in other cases, for example, as heuristics for the TSP and vehicle routing with time windows.

We stated in Corollary 11.3 that we are able to construct the reduced MDD $\mathscr{M}$ when only precedence constraints are imposed and a sufficiently large W is given. We have the following results for $\mathscr{M}$ if the precedence relations satisfy the discrepancy structure for a given k:

Lemma 11.5. *We have* $All_v^{\downarrow} \subseteq \{j_1,\ldots,j_{\min\{m+k-1,n\}}\}$ *for any given node* $v \in L_{m+1}$, $m = 1,\ldots,n$.

Proof. If $m+k-1 > n$ we obtain the redundant condition $All_u^{\downarrow} \subseteq \mathscr{J}$, therefore assume $m+k-1 \leq n$. Suppose there exists $j_l \in All_v^{\downarrow}$ for some $v \in L_{m+1}$ such that $l > m+k-1$. Then, for any $i = 1,\ldots,m$, we have $l-i \geq m+k-i \geq m+k-m = k$. This implies $\{j_1,\ldots,j_m\} \subset All_v^{\downarrow}$, since job j_l belongs to a partial ordering π only if all jobs j_i for which $l-i \geq k$ are already accounted for in π. But then $|All_v^{\downarrow}| \geq m+1$, which is a contradiction since $v \in L_{m+1}$ implies that $|All_v^{\downarrow}| = m$, as any partial ordering identified by a path from $\mathbf{r}$ to v must contain m distinct jobs. □

Theorem 11.2. *The width of* $\mathscr{M}$ *is* 2^{k-1}.

Proof. Let us first assume $n \geq k+2$ and restrict our attention to layer L_{m+1} for some $m \in \{k,\ldots,n-k+1\}$. Also, let $\mathscr{F} := \{All_u^{\downarrow} : u \in L_{m+1}\}$. It can be shown that, if $\mathscr{M}$ is reduced, no two nodes $u, v \in L_{m+1}$ are such that $All_u^{\downarrow} = All_v^{\downarrow}$. Thus, $|\mathscr{F}| = |L_{m+1}|$.

We derive the cardinality of $\mathscr{F}$ as follows: Take $All_v^{\downarrow} \in \mathscr{F}$ for some $v \in L_{m+1}$. Since $|All_v^{\downarrow}| = m$, there exists at least one job $j_i \in All_v^{\downarrow}$ such that $i \geq m$. According to Lemma 11.5, the maximum index of a job in $All_v^{\downarrow}$ is $m+k-1$. So consider the jobs indexed by $m+k-1-l$ for $l = 0,\ldots,k-1$; at least one of them is necessarily contained in $All_v^{\downarrow}$. Due to the discrepancy precedence constraints, $j_{m+k-1-l} \in All_v^{\downarrow}$ implies that any j_i with $i \leq m-l-1$ is also contained in $All_v^{\downarrow}$ (if $m-l-1 > 0$).

Now, consider the sets in $\mathscr{F}$ which contain a job with index $m+k-1-l$, but do *not* contain any job with index greater than $m+k-1-l$. Any such set $All_u^{\downarrow}$ contains the jobs $j_1,\dots,j_{m-l-1}$ according to Lemma 11.5. Hence, the remaining $m-(m-l-1)-1=l$ job indices can be freely chosen from the sequence $m-l,\dots,m+k-l-2$. Notice there are no imposed precedences on these remaining $m+k-l-2-(m-l)+1=k-1$ elements; thus, there exist $\binom{k-1}{l}$ such subsets. But these sets define a partition of $\mathscr{F}$. Therefore

$$|\mathscr{F}| = |L_{m+1}| = \sum_{l=0}^{k-1} \binom{k-1}{l} = \binom{k-1}{0} + \cdots + \binom{k-1}{k-1} = 2^{k-1}.$$

We can use an analogous argument for the layers L_{m+1} such that $m<k$ or $m>n-k+1$, or when $k=n-1$. The main technical difference is that we have fewer than $k-1$ possibilities for the new combinations, and so the maximum number of nodes is strictly less than 2^{k-1} for these cases. Thus the width of $\mathscr{M}$ is 2^{k-1}. □

According to Theorem 11.2, $\mathscr{M}$ has $O(n\,2^{k-1})$ nodes as it contains $n+1$ layers. Since arcs only connect nodes in adjacent layers, the MDD contains $O(n\,2^{2k-2})$ arcs (assuming a worst-case scenario where all nodes in a layer are adjacent to all nodes in the next layer, yielding at most $2^{k-1}\cdot 2^{k-1}=2^{2k-2}$ arcs directed out of a layer). Using the recursive relation (11.2) in Section 11.3, we can compute, e.g., the minimum sum of setup times in worst-case time complexity of $O(n^2\,2^{2k-2})$. The work by [14] provides an algorithm that minimizes this same function in $O(nk^2\,2^{k-2})$, but that is restricted to this particular objective.

11.9 Application to Constraint-Based Scheduling

We next describe how the techniques of the previous sections can be added to IBM ILOG CP Optimizer (CPO), a state-of-the-art general-purpose constraint programming solver. In particular, it contains dedicated syntax and associated propagation algorithms for sequencing and scheduling problems. Given a sequencing problem as considered in this chapter, CPO applies a depth-first branch-and-bound search where jobs are recursively appended to the end of a partial ordering until no jobs are left unsequenced. At each node of the branching tree, a number of sophisticated propagation algorithms are used to reduce the possible candidate jobs to be appended to

the ordering. Examples of such propagators include *edge-finding*, *not-first/not-last rules*, and *deductible precedences*; details can be found in [17] and [154].

We have implemented our techniques by introducing a new user-defined constraint type to the CPO system, representing a generic sequencing problem. We maintain a relaxed MDD for this constraint type, and we implemented the filtering and refinement techniques in the associated propagation algorithm. The constraint participates in the constraint propagation cycle of CPO; each time it is activated it runs one round of top-down filtering and refinement. In particular, the filtering operation takes into account the search decisions up to that point (i.e., the jobs that are already fixed in the partial ordering) and possible precedence constraints that are deduced by CPO. At the end of a round, we use the relaxed MDD to reduce the number of candidate successor jobs (by analyzing the arc labels in the appropriate layers) and to communicate new precedence constraints as described in Section 11.6, which may trigger additional propagation by CPO. Our implementation follows the guidelines from [101].

In this section we present computational results for different variations of single-machine sequencing problems using the MDD-based propagator. Our goal is twofold. First, we want to analyze the sensitivity of the relaxed MDD with respect to the width and refinement strategy. Second, we wish to provide experimental evidence that combining a relaxed MDD with existing techniques for sequencing problems can improve the performance of constraint-based solvers.

11.9.1 Experimental Setup

Three formulations were considered for each problem: a CPO model with its default propagators, denoted by `CPO`; a CPO model containing only the MDD-based propagator, denoted by `MDD`; and a CPO model with the default and MDD-based propagators combined, denoted by `CPO+MDD`. The experiments mainly focus on the comparison between `CPO` and `CPO+MDD`, as these indicate whether incorporating the MDD-based propagator can enhance existing methods.

We have considered two heuristic strategies for selecting the next job to be appended to a partial schedule. The first, denoted by *lex search*, is a static method that always tries to first sequence the job with the smallest index, where the index of a job is fixed per instance and defined by the order in which it appears in the input. This allows for a more accurate comparison between two propagation methods,

since the branching tree is fixed. In the second strategy, denoted by *dynamic search*, the CPO engine automatically selects the next job according to its own state-of-the-art scheduling heuristics. The purpose of the experiments that use this search is to verify how the MDD-based propagator is influenced by strategies that are known to be effective for constraint-based solvers. The dynamic search is only applicable to `CPO` and `CPO+MDD`.

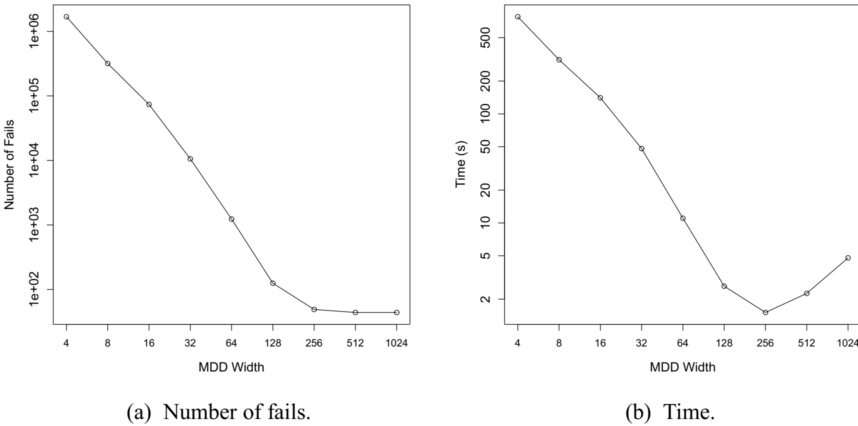
We measure two performance indicators: the total solving time and the *number of fails*. The number of fails corresponds to the number of times during search that a partial ordering was detected to be infeasible, i.e., either some constraint is violated or the objective function is greater than a known upper bound. The number of fails is proportional to the size of the branching tree and, hence, to the total solving time of a particular technique.

The techniques presented here do not explore any additional problem structure that was not described in this chapter, such as specific search heuristics, problem relaxations, or dominance criteria (except only if such structure is already explored by CPO). More specifically, we used the same MDD-based propagator for all problems, which dynamically determines what node state and refinement strategy to use according to the input constraints and the objective function.

The experiments were performed on a computer equipped with an Intel Xeon E5345 at 2.33 GHz with 8 GB RAM. The MDD code was implemented in C++ using the CPO callable library from ILOG CPLEX Academic Studio V.12.4.01. We set the following additional CPO parameters for all experiments: `Workers=1`, to use a single computer core; `DefaultInferenceLevel=Extended`, to use the maximum possible propagation available in CPO; and `SearchType=DepthFirst`.

11.9.2 Impact of the MDD Parameters

We first investigate the impact of the maximum width and refinement on the number of fails and total solving time for the MDD approaches. As a representative test case, we consider the traveling salesman problem with time windows (TSPTW). The TSPTW is the problem of finding a minimum-cost tour in a weighted digraph starting from a selected vertex (the depot), visiting each vertex within a given time window, and returning to the original vertex. In our case, each vertex is a job, the release dates and deadlines are defined according to the vertex time windows, and



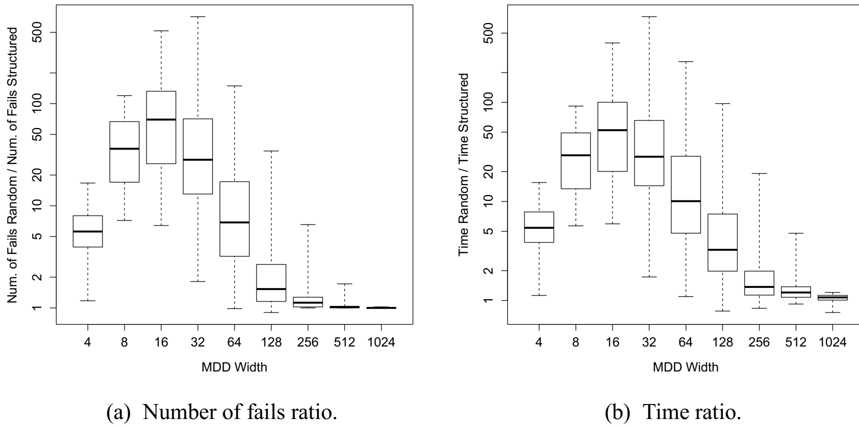
(a) Number of fails. (b) Time.

Fig. 11.4 Impact of the MDD width on the number of fails and total time for the TSPTW instance `n20w200.001` from the Gendreau class. The axes are in logarithmic scale.

travel distances are perceived as setup times. The objective function is to minimize the sum of setup times.

We selected the instance `n20w200.001` from the well-known Gendreau benchmark proposed by [72], as it represents the typical behavior of an MDD. It consists of a 20-vertex graph with an average time window width of 200 units. The tested approach was the `MDD` model with lex search. We used the following job ranking for the refinement strategy described in Section 11.7: The first job in the ranking, j_1^*, was set as the first job of the input. The i-th job in the ranking, j_i^*, is the one that maximizes the sum of the setup times to the jobs already ranked, i.e., $j_i^* = \arg\max_{p \in \mathscr{J} \setminus \{j_1^*, \ldots, j_{i-1}^*\}} \{\sum_{k=1}^{i-1} t_{j_k^*, p}\}$ for the setup times t. The intuition is that we want jobs with largest travel distances to be exactly represented in $\mathscr{M}$.

The number of fails and total time to find the optimal solution for different MDD widths are presented in Fig. 11.4. Due to the properties of the refinement technique in Theorem 4.3, we consider only powers of 2 as widths. We note from Fig. 11.4(a) that the number of fails is decreasing rapidly as the width increases, up to a point where it becomes close to constant (from 512 to 1024). This indicates that, at a certain point, the relaxed MDD is very close to an actual exact representation of the problem, and hence no benefit is gained from any increment of the width. The number of fails has a direct impact on the total solving time, as observed in Fig. 11.4(b). Namely, the times decrease accordingly as the width increases. At



(a) Number of fails ratio.

(b) Time ratio.

Fig. 11.5 Performance comparison between random and structured refinement strategies for the TSPTW instance `n20w200.001`. The axes are in logarithm scale.

the point where the relaxed MDD is close to exact, larger widths only introduce additional overhead, thus increasing the solving time.

To analyze the impact of the refinement, we generated 50 job rankings uniformly at random for the refinement strategy described in Section 11.7. These rankings were compared with the *structured* one for setup times used in the previous experiment. To make this comparison, we solved the `MDD` model with lex search for each of the 51 refinement orderings, considering widths from 4 to 1024. For each random order, we divided the resulting number of fails and time by the ones obtained with the structured refinement for the same width. Thus, this ratio represents how much better the structured refinement is over the random strategies. The results are presented in the box-and-whisker plots of Fig. 11.5. For each width the horizontal lines represent, from top to bottom, the maximum observed ratio, the upper quartile, the median ratio, the lower quartile, and the minimum ratio.

We interpret Fig. 11.5 as follows: An MDD with very small width captures little of the jobs that play a more important role in the optimality or feasibility of the problem, in view of Theorem 4.3. Thus, distinct refinement strategies are not expected to differ much on average, as shown, e.g., in the width-4 case of Fig. 11.5(a). As the width increases, there is a higher chance that these crucial jobs are better represented by the MDD, leading to a good relaxation, but also a higher chance that little of their structure is captured by a random strategy, leading in turn to a weak relaxation. This yields a larger variance in the refinement performance.

Finally, for sufficiently large widths, we end up with an almost exact representation of the problem and the propagation is independent of the refinement order (e.g., widths 512 and 1024 of Fig. 11.5(a)). Another aspect we observe in Fig. 11.5(b) is that, even for relatively small widths, the structured refinement can be orders of magnitude better than a random one. This emphasizes the importance of applying an appropriate refinement strategy for the problem at hand.

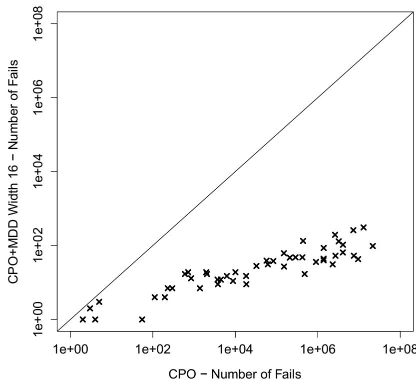
11.9.3 Traveling Salesman Problem with Time Windows

We first evaluate the relative performance of `CPO` and `CPO+MDD` on sequencing problems with time window constraints, and where the objective is to minimize the sum of setup times. We considered a set of well-known TSPTW instances defined by the Gendreau, Dumas, and Ascheuer benchmark classes, which were proposed by [72], [56], and [10], respectively. We selected all instances with up to 100 jobs, yielding 388 test cases in total. The `CPO` and the `CPO+MDD` models were initially solved with lex search, considering a maximum width of 16. A time limit of 1,800 seconds was imposed for all methods, and we used the structured job ranking described in Section 11.9.2.

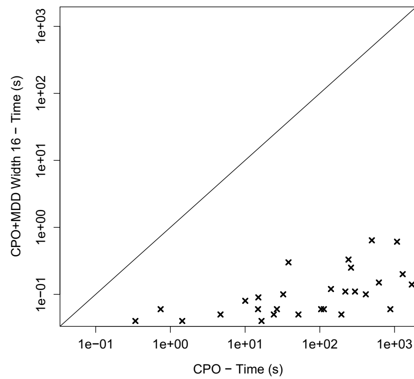
The `CPO` approach was able to solve 26 instances to optimality, while the `CPO+MDD` approach solved 105 instances to optimality. The number of fails and solution times are presented in the scatter plots of Fig. 11.6, where we only considered instances solved by both methods. The plots provide a strong indication that the MDD-based propagator can greatly enhance the CPO inference mechanism. For example, `CPO+MDD` can reduce the number of fails from over 10 million (`CPO`) to less than 100 for some instances.

In our next experiment we compared `CPO` and `CPO+MDD` considering a maximum width of 1024 and applying instead a dynamic search, so as to verify if we could still obtain additional gains with the general-purpose scheduling heuristics provided by CPO. A time limit of 1,800 seconds was imposed for all approaches.

With the above configuration, the `CPO` approach solved to optimality 184 out of the 388 instances, while the `CPO+MDD` approach solved to optimality 311 instances. Figure 11.7(a) compares the times for instances solved by both methods, while Fig. 11.7(b) depicts the performance plot. In particular, the overhead introduced by the MDD is only considerable for small instances (up to 20 jobs). In the majority of the cases, `CPO+MDD` is capable of proving optimality much quicker.

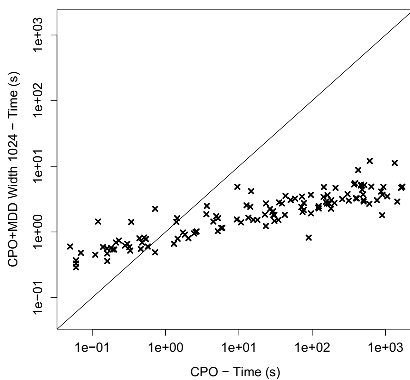


(a) Number of fails.

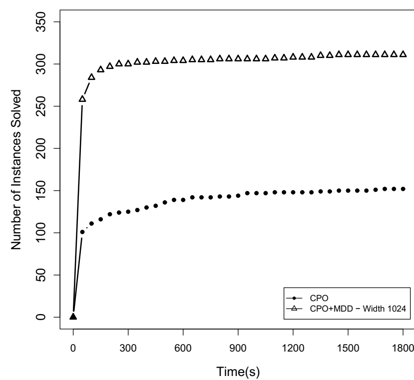


(b) Time.

Fig. 11.6 Performance comparison between CPO and CPO+MDD for minimizing sum of setup times on Dumas, Gendreau, and Ascheuer TSPTW classes with lex search. The vertical and horizontal axes are in logarithmic scale.



(a) Scatter plot.



(b) Performance plot.

Fig. 11.7 Performance comparison between CPO and CPO+MDD for minimizing sum of setup times on Dumas, Gendreau, and Ascheuer TSPTW classes using default depth-first CPO search. The horizontal and vertical axes in (a) are in logarithmic scale.

11.9.4 Asymmetric Traveling Salesman Problem with Precedence Constraints

We next evaluate the performance of CPO and CPO+MDD on sequencing problems with precedence constraints, while the objective is again to minimize the sum of

setup times. As benchmark problem, we consider the asymmetric traveling salesman problem with precedence constraints (ATSPP), also known as the *sequential ordering problem*. The ATSPP is a variation of the asymmetric TSP where precedence constraints must be observed. Namely, given a weighted digraph $D = (V,A)$ and a set of pairs $P = V \times V$, the ATSPP is the problem of finding a minimum-weight Hamiltonian tour T such that vertex v precedes u in T if $(v,u) \in P$.

The ATSPP has been shown to be extremely challenging for exact methods. In particular, a number of instances with fewer than 70 vertices from the well-known [147] benchmark, proposed initially by [11], are still open. We refer to the work of [6] for a more detailed literature review of exact and heuristic methods for the ATSPP.

We applied the `CPO` and `CPO+MDD` models with dynamic search and a maximum width of 2048 for 16 instances of the ATSPP from the TSPLIB benchmark. A time limit of 1,800 seconds was imposed, and we used the structured job ranking described in Section 11.9.2. The results are reported in Table 11.1. For each instance we report the size (number of vertices) and the current best lower and upper bound from the literature.[1] The column 'Best' corresponds to the best solution found by a method, and the column 'Time' corresponds to the computation time in which the solution was proved optimal. A value TL indicates that the time limit was reached.

We were able to close three of the unsolved instances with our generic approach, namely p43.2, p43.3, and ry48p.4. In addition, instance p43.4 was solved before with more than 22 hours of CPU time by [92] (for a computer approximately 10 times slower than ours), and by more than 4 hours by [76] (for an unspecified machine), while we could solve it in less than 90 seconds. The presence of more precedence constraints (indicated for these instances by a larger suffix number) is more advantageous to our MDD approach, as shown in Table 11.1. On the other hand, less constrained instances are better suited to approaches based on mixed integer linear programming; instances p43.1 and ry48p.1 are solved by a few seconds in [11].

As a final observation, we note that the bounds for the p43.1-4 instances reported in the TSPLIB are inconsistent. They do not match any of the bounds from existing works we are aware of or the ones provided by [11], where these problems were proposed. This includes the instance p43.1 which was solved in that work.

---

[1] Since the TSPLIB results are not updated on the TSPLIB website, we report updated bounds obtained from [92], [76], and [6].

Table 11.1 Results on ATSPP instances. Values in bold represent instances solved for the first time. TL represents that the time limit (1,800 s) was reached.

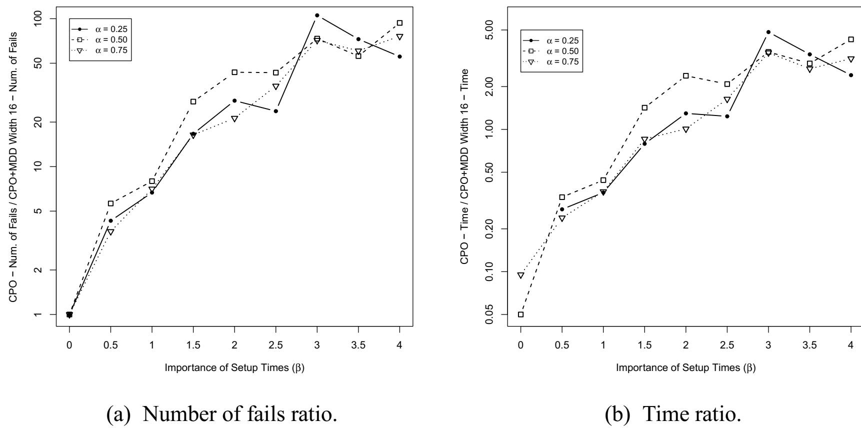
Instance	Vertices	Bounds	CPO		CPO+MDD width 2048	
			Best	Time (s)	Best	Time (s)
br17.10	17	55	55	0.01	55	4.98
br17.12	17	55	55	0.01	55	4.56
ESC07	7	2125	2125	0.01	2125	0.07
ESC25	25	1681	1681	TL	1681	48.42
p43.1	43	28140	28205	TL	28140	287.57
p43.2	43	[28175, 28480]	28545	TL	**28480**	**279.18**
p43.3	43	[28366, 28835]	28930	TL	**28835**	**177.29**
p43.4	43	83005	83615	TL	83005	88.45
ry48p.1	48	[15220, 15805]	18209	TL	16561	TL
ry48p.2	48	[15524, 16666]	18649	TL	17680	TL
ry48p.3	48	[18156, 19894]	23268	TL	22311	TL
ry48p.4	48	[29967, 31446]	34502	TL	**31446**	**96.91**
ft53.1	53	[7438, 7531]	9716	TL	9216	TL
ft53.2	53	[7630, 8026]	11669	TL	11484	TL
ft53.3	53	[9473, 10262]	12343	TL	11937	TL
ft53.4	53	14425	16018	TL	14425	120.79

11.9.5 Makespan Problems

Constraint-based solvers are known to be particularly effective when the objective is to minimize makespan, which is largely due to specialized domain propagation techniques that can be used in such cases; see, e.g., [17].

In this section we evaluate the performance of CPO and CPO+MDD on sequencing problems with time window constraints and where the objective is to minimize makespan. Our goal is to test the performance of such procedures on makespan problems, and verify the influence of setup times on the relative performance. In particular, we will empirically show that the MDD-based propagator makes schedulers more robust for makespan problems, especially when setup times are present.

To compare the impact of setup times between methods, we performed the following experiment: Using the scheme from [41], we first generated three random instances with 15 jobs. The processing times p_i are selected uniformly at random from the set $\{1,100\}$, and release dates are selected uniformly at random from the set $\{0,\ldots,\alpha\sum_i p_i\}$ for $\alpha \in \{0.25, 0.5, 0.75\}$. No deadlines are considered. For each of the three instances above, we generated additional random instances where we add a setup time for all pairs of jobs i and j selected uniformly at random from the set



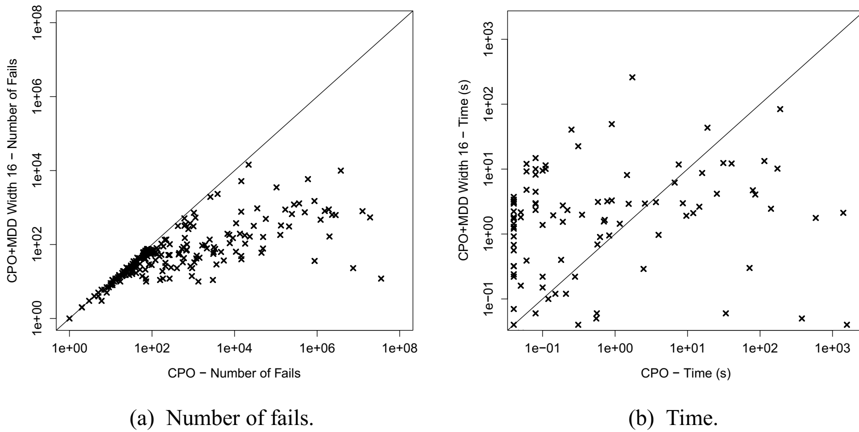
(a) Number of fails ratio. (b) Time ratio.

Fig. 11.8 Comparison between CPO and CPO+MDD for minimizing makespan on three instances with randomly generated setup times. The vertical axes are in logarithmic scale.

$\{0,\ldots,(50.5)\beta\}$, where $\beta \in \{0,0.5,1,\ldots,4\}$. In total, 10 instances are generated for each β. We computed the number of fails and total time to minimize the makespan using CPO and CPO+MDD models with a maximum width of 16, applying a lex search in both cases. We then divided the CPO results by the CPO+MDD results, and computed the average ratio for each value of β. The job ranking for refinement is done by sorting the jobs in decreasing order according to the value obtained by summing their release dates with their processing times. This forces jobs with larger completion times to have higher priority in the refinement.

The results are presented in Fig. 11.8. For each value of α, we plot the ratio of CPO and CPO+MDD in terms of the number of fails (Fig. 11.8(a)) and time (Fig. 11.8(b)). The plot in Fig. 11.8(a) indicates that the CPO+MDD inference becomes more dominant in comparison with CPO for larger values of β, that is, when setup times become more important. The MDD introduces a computational overhead in comparison with the CPO times (around 20 times slower for this particular problem size). This is compensated as β increases, since the number of fails for the CPO+MDD model becomes orders of magnitude smaller in comparison with CPO. The same behavior was observed on average for other base instances generated under the same scheme.

To evaluate this on structured instances, we consider the TSPTW instances defined by the Gendreau and Dumas benchmark classes, where we changed the objective function to minimize makespan instead of the sum of setup times. We



(a) Number of fails. (b) Time.

Fig. 11.9 Performance comparison between CPO and CPO+MDD for minimizing makespan on Dumas and Gendreau TSPTW classes. The vertical and horizontal axes are in logarithmic scale.

selected all instances with up to 100 jobs, yielding 240 test cases in total. We solved the CPO and the CPO+MDD models with lex search, so as to compare the inference strength for these problems. A maximum width of 16 was set for CPO+MDD, and a time limit of 1,800 seconds was imposed for both cases. The job ranking is the same as in the previous experiment.

The CPO approach was able to solve 211 instances to optimality, while the CPO+MDD approach solved 227 instances to optimality (including all the instances solved by CPO). The number of fails and solving time are presented in Fig. 11.9, where we only depict instances solved by both methods. In general, for easy instances (up to 40 jobs or with a small time window width), the reduction of the number of fails induced by CPO+MDD was not significant, and thus did not compensate the computational overhead introduced by the MDD. However, we note that the MDD presented better performance for harder instances; the lower diagonal of Fig. 11.9(b) is mostly composed by instances from the Gendreau class with larger time windows, for which the number of fails was reduced by five and six orders of magnitude. We also note that the result for the makespan objective is less pronounced than for the sum of setup times presented in Section 11.9.3.

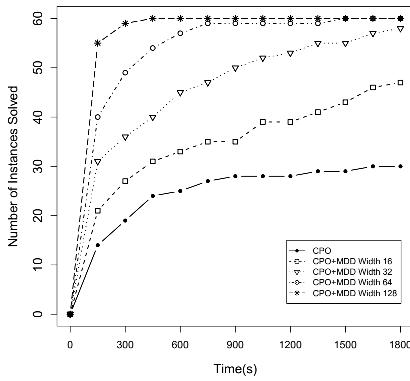
11.9.6 Total Tardiness

Constraint-based schedulers are usually equipped with specific filtering techniques for minimizing total tardiness, which are based on the propagation of a piecewise-linear function as described by [17]. For problems without any constraints, however, the existing schedulers are only capable of solving small instances, and heuristics end up being more appropriate as the propagators are not sufficiently strong to deduce good bounds.

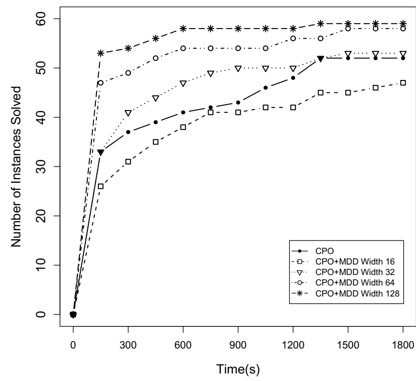
In this section we evaluate the performance of CPO and CPO+MDD on sequencing problems where the objective is to minimize the total tardiness. Since we are interested in evaluating the inference strength of the objective function bounding mechanism, we do not take into account any additional side constraints and we limit our problem size to 15 jobs. Moreover, jobs are only subject to a release date, and no setup time is considered.

We have tested the total tardiness objective using random instances, again generated with the scheme of [41]. The processing times p_i are selected uniformly at random from the set $\{1,10\}$, the release dates r_i are selected uniformly at random from the set $\{0,\dots,\alpha\sum_i p_i\}$, and the due dates are selected uniformly at random from the set $\{r_i+p_i,\dots,r_i+p_i+\beta\sum_i p_i\}$. To generate a good diversity of instances, we considered $\alpha \in \{0,0.5,1.0,1.5\}$ and $\beta \in \{0.05,0.25,0.5\}$. For each random instance generated, we create a new one with the same parameters but where we assign tardiness weights selected uniformly at random from the set $\{1,\dots,10\}$. We generated 5 instances for each configuration, hence 120 instances in total. A time limit of 1,800 seconds was imposed for all methods. The ranking procedure for refinement is based on sorting the jobs in decreasing order of their due dates.

We compared the CPO and the CPO+MDD models for different maximum widths, and lex search was applied to solve the models. The results for unweighted total tardiness are presented in Fig. 11.10(a), and the results for the weighted total tardiness instances are presented in Fig. 11.10(b). We observe that, even for relatively small widths, the CPO+MDD approach was more robust than CPO for unweighted total tardiness; more instances were solved in less time even for a width of 16, which is a reflection of a great reduction of the number of fails. On the other hand, for weighted total tardiness CPO+MDD required larger maximum widths to provide a more significant benefit with respect to CPO. We believe that this behavior may be due to a weaker refinement for the weighted case, which may require larger widths to capture the set of activities that play a bigger role in the final solution cost.



(a) Total tardiness.



(b) Weighted total tardiness.

Fig. 11.10 Performance comparison between `CPO` and `CPO+MDD` for minimizing total tardiness on randomly generated instances with 15 jobs.

In all cases, a minimum width of 128 would suffice for the MDD propagation to provide enough inference to solve all the considered problems.

References

[1] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, New York, 1997.

[2] H. Abeledo, R. Fukasawa, A. Pessoa, and E. Uchoa. The time dependent traveling salesman problem: polyhedra and algorithm. *Mathematical Programming Computation*, 5(1):27–55, 2013.

[3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[4] H. R. Andersen, T. Hadžić, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.

[5] H. R. Andersen, T. Hadžić, and D. Pisinger. Interactive cost configuration over decision diagrams. *Journal of Artificial Intelligence Research*, 37:99–139, 2010.

[6] D. Anghinolfi, R. Montemanni, M. Paolucci, and L. M. Gambardella. A hybrid particle swarm optimization approach for the sequential ordering problem. *Computers & Operations Research*, 38(7):1076–1085, 2011.

[7] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[8] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a *k*-tree. *SIAM Journal on Algebraic and Discrete Mathematics*, 8:277–284, 1987.

[9] S. Arnborg and A. Proskurowski. Characterization and recognition of partial *k*-trees. *SIAM Journal on Algebraic and Discrete Mathematics*, 7:305–314, 1986.

[10] N. Ascheuer. *Hamiltonian Path Problems in the On-line Optimization of Flexible Manufacturing Systems*. PhD thesis, Technische Universität Berlin, Germany, 1995.

[11] N. Ascheuer, M. Jünger, and G. Reinelt. A branch and cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17:61–84, 2000.

[12] K. R. Baker and B. Keller. Solving the single-machine sequencing problem using integer programming. *Computers and Industrial Engineering*, 59:730–735, 2010.

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9

[13] E. Balas. A linear characterization of permutation vectors. Management science research report 364, Carnegie Mellon University, 1975.

[14] E. Balas. New classes of efficiently solvable generalized traveling salesman problems. *Annals of Operations Research*, 86:529–558, 1999.

[15] E. Balas and N. Simonetti. Linear time dynamic-programming algorithms for new classes of restricted TSPs: A computational study. *INFORMS Journal on Computing*, 13:56–75, December 2000.

[16] B. Balasundaram, S. Butenko, and I. V. Hicks. Clique relaxations in social network analysis: The maximum *k*-plex problem. *Operations Research*, 59(1):133–142, January 2011.

[17] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer, 2001.

[18] B. Becker, M. Behle, F. Eisenbrand, and R. Wimmer. BDDs in a branch and cut framework. In S. Nikoletseas, editor, *Proceedings, International Workshop on Efficient and Experimental Algorithms (WEA)*, volume 3503 of *LNCS*, pages 452–463. Springer, 2005.

[19] M. Behle. *Binary Decision Diagrams and Integer Programming*. PhD thesis, Max Planck Institute for Computer Science, 2007.

[20] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 107–122. Springer, 2004.

[21] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[22] D. Bergman. *New Techniques for Discrete Optimization*. PhD thesis, Tepper School of Business, Carnegie Mellon University, 2013.

[23] D. Bergman, A. Ciré, W.-J. van Hoeve, and J. N. Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26(2):253–268, 2014.

[24] D. Bergman, A. A. Ciré, W.-J. van Hoeve, and J. N. Hooker. Variable ordering for the application of BDDs to the maximum independent set problem. In *CPAIOR Proceedings*, volume 7298 of *LNCS*, pages 34–49. Springer, 2012.

[25] D. Bergman, A. A. Ciré, W.-J. van Hoeve, and J. N. Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26:253–268, 2013.

[26] D. Bergman, A. A. Ciré, W.-J. van Hoeve, and J. N. Hooker. Discrete optimization with binary decision diagrams. *INFORMS Journal on Computing*, 28:47–66, 2016.

[27] D. Bergman, A. A. Ciré, W.-J. van Hoeve, and T. Yunes. BDD-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, 2014.

[28] D. Bergman, W.-J. van Hoeve, and J. N. Hooker. Manipulating MDD relaxations for combinatorial optimization. In T. Achterberg and C. Beck, editors, *CPAIOR Proceedings*, volume 6697 of *LNCS*. Springer, 2011.

[29] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, New York, 1972.

[30] T. Berthold. Primal heuristics for mixed integer programs. Master's thesis, Zuse Institute Berlin, 2006.

[31] D. Bertsimas, D. A. Iancu, and D. Katz. A new local search algorithm for binary optimization. *INFORMS Journal on Computing*, 25(2):208–221, 2013.

[32] C. Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 29–83. Elsevier, 2006.

[33] R. E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica. Extra Volume: Optimization Stories*, pages 107–121, 2012.

[34] B. Bloom, D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. Saraswat. SatX10: A scalable plug & play parallel SAT framework. In *SAT Proceedings*, volume 7317 of *LNCS*, pages 463–468. Springer, 2012.

[35] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45:993–1002, 1996.

[36] S. Brand, N. Narodytska, C.G. Quimper, P. Stuckey, and T. Walsh. Encodings of the sequence constraint. In *Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *LNCS*, pages 210–224. Springer, 2007.

[37] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[38] N. J. Calkin and H. S. Wilf. The number of independent sets in a grid graph. *SIAM Journal on Discrete Mathematics*, 11(1):54–60, 1998.

[39] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:2000, 1998.

[40] A. Caprara, A. N. Letchford, and J-J. Salazar-González. Decorous lower bounds for minimum linear arrangement. *INFORMS Journal on Computing*, 23:26–40, 2011.

[41] S. Chang, Q. Lu, G. Tang, and W. Yu. On decomposition of the total tardiness problem. *Operations Research Letters*, 17(5):221 – 229, 1995.

[42] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings, Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 519–538, San Diego, 2005.

[43] K. C. K. Cheng, W. Xia, and R. H. C. Yap. Space-time tradeoffs for the regular constraint. In *Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *LNCS*, pages 223–237. Springer, 2012.

[44] K. C. K. Cheng and R. H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In G. Brewka et al., editor, *Proceedings of ECAI*, pages 78–82. IOS Press, 2006.

[45] K. C. K. Cheng and R. H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *LNCS*, pages 509–523. Springer, 2008.

[46] D. Chhajed and T. J. Lowe. Solving structured multifacility location problems efficiently. *Transportation Science*, 28:104–115, 1994.

[47] N. Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

[48] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Principles and Practice of Constraint Programming (CP 2009)*, volume 5732 of *LNCS*, pages 226–241, 2009.

[49] A. A. Ciré and W.-J. van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

[50] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[51] G. M. Del Corso and G. Manzini. Finding exact solutions to the bandwidth minimization problem. *Computing*, 62(3):189–203, 1999.

[52] Y. Crama, P. Hansen, and B. Jaumard. The basic algorithm for pseudoboolean programming revisited. *Discrete Applied Mathematics*, 29:171–185, 1990.

[53] J. Cussens. Bayesian network learning by compiling to weighted MAX-SAT. In *Proceedings, Uncertainty in Artificial Intelligence (UAI)*, pages 105–112, Helsinki, 2008.

[54] R. Dechter. Bucket elimination: A unifying framework for several probabilistic inference algorithms. In *Proceedings, Uncertainty in Artificial Intelligence (UAI)*, pages 211–219, Portland, OR, 1996.

[55] N. Downing, T. Feydy, and P. Stuckey. Explaining flow-based propagation. In *CPAIOR Proceedings*, volume 7298 of *LNCS*, pages 146–162. Springer, 2012.

[56] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.

[57] M. E. Dyer, A. M. Frieze, and M. Jerrum. On counting independent sets in sparse graphs. *SIAM Journal on Computing*, 31(5):1527–1541, 2002.

[58] R. Ebendt, W. Gunther, and R. Drechsler. An improved branch and bound algorithm for exact BDD minimization. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 22(12):1657–1663, 2003.

[59] J. D. Eblen, C. A. Phillips, G. L. Rogers, and M. A. Langston. The maximum clique enumeration problem: Algorithms, applications and implementations. In *Proceedings, International Symposium on Bioinformatics Research and Applications (ISBRA)*, pages 306–319. Springer, 2011.

[60] J. Eckstein and M. Nediak. Pivot, cut, and dive: A heuristic for 0–1 mixed integer programming. *Journal of Heuristics*, 13(5):471–503, 2007.

[61] J. Edachery, A. Sen, and F. J. Brandenburg. Graph clustering using distance-k cliques. In *Proceedings of Graph Drawing*, volume 1731 of *LNCS*, pages 98–106. Springer, 1999.

[62] U. Feige. Approximating the bandwidth via volume respecting embeddings. *Journal of Computer Systems Science*, 60(3):510–539, 2000.

[63] Z. Feng, E. A. Hansen, and S. Zilberstein. Symbolic generalization for on-line planning. In *Proceedings, Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 109–116. Morgan Kaufmann, 2003.

[64] P. Festa, P. M. Pardalos, M. G. C. Resende, and C. C. Ribeiro. Randomized heuristics for the max-cut problem. *Optimization Methods and Software*, 7:1033–1058, 2002.

[65] S. Fiorini, S. Massar, S. Pokutta, H. R. Tiwary, and R. de Wolf. Linear vs. semidefinite extended formulations: Exponential separation and strong lower

bounds. In *Proceedings, ACM Symposium on Theory of Computing (STOC)*, pages 95–106, New York, 2012. ACM.

[66] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

[67] F. Forbes and B. Ycart. Counting stable sets on Cartesian products of graphs. *Discrete Mathematics*, 186(1-3):105–116, 1998.

[68] G. Freuder and M. Wallace. Constraint technology and the commercial world. *Intelligent Systems and their Applications, IEEE*, 15(1):20–23, 2000.

[69] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[70] G. Gange, V. Lagoon, and P. J. Stuckey. Fast set bounds propagation using BDDs. In M. Ghallab et al., editor, *Proceedings, European Conference on Artificial Intelligence (ECAI)*, pages 505–509. IOS Press, 2008.

[71] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[72] M. Gendreau, A. Hertz G., Laporte, and M. Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, March 1998.

[73] F. Glover and M. Laguna. General purpose heuristics for integer programming – Part I. *Journal of Heuristics*, 2(4):343–358, 1997.

[74] F. Glover and M. Laguna. General purpose heuristics for integer programming – Part II. *Journal of Heuristics*, 3(2):161–179, 1997.

[75] C. P. Gomes, C. Fernández, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337, 2005.

[76] L. Gouveia and P. Pesneau. On extended formulations for the precedence constrained asymmetric traveling salesman problem. *Networks*, 48(2):77–89, 2006.

[77] A. Grosso, M. Locatelli, and W. Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *Journal of Heuristics*, 14(6):587–612, 2008.

[78] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2. Springer, 1993.

[79] Z. Gu. Gurobi optimization—Gurobi compute server, distributed tuning tool and distributed concurrent MIP solver. In *INFORMS Annual Meeting*, 2013. See also http://www.gurobi.com/products/gurobi-compute-server/distributed-optimization.

[80] E. M. Gurari and I. H. Sudborough. Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. *Journal of Algorithms*, 5:531–546, 1984.

[81] G. D. Hachtel and F. Somenzi. A symbolic algorithm for maximum flow in 0–1 networks. *Formal Methods in System Design*, 10(2-3):207–219, April 1997.

[82] T. Hadžić and J. N. Hooker. Discrete global optimization with binary decision diagrams. In *Workshop on Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry (GICOLAG)*, Vienna, 2006.

[83] T. Hadžić and J. N. Hooker. Cost-bounded binary decision diagrams for 0–1 programming. In E. Loute and L. Wolsey, editors, *CPAIOR Proceedings*, volume 4510 of *LNCS*, pages 84–98. Springer, 2007.

[84] T. Hadžić, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *LNCS*, pages 448–462. Springer, 2008.

[85] T. Hadžić, J. N. Hooker, and P. Tiedemann. Propagating separable equalities in an MDD store. In L. Perron and M. A. Trick, editors, *CPAIOR Proceedings*, volume 5015 of *LNCS*, pages 318–322. Springer, 2008.

[86] T. Hadžić and J.N. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University, 2006.

[87] T. Hadžić, E. O'Mahony, B. O'Sullivan, and M. Sellmann. Enhanced inference for the market split problem. In *Proceedings, International Conference on Tools for AI (ICTAI)*, pages 716–723. IEEE, 2009.

[88] W. W. Hager and Y. Krylyuk. Graph partitioning and continuous quadratic programming. *SIAM Journal on Discrete Mathematics*, 12(4):500–523, 1999.

[89] U.-U. Haus and C. Michini. Representations of all solutions of Boolean programming problems. In *Proceedings, International Symposium on AI and Mathematics (ISAIM)*, 2014.

[90] P. Hawkins, V. Lagoon, and P.J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24(1):109–156, 2005.

[91] C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization*, 10:673–696, 1997.

[92] I. T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. In *Operations Research Proceedings*, pages 355–362. Springer, 2003.

[93] S. Hoda. *Essays on Equilibrium Computation, MDD-based Constraint Programming and Scheduling*. PhD thesis, Carnegie Mellon University, 2010.

[94] S. Hoda, W.-J. van Hoeve, and J. N. Hooker. A systematic approach to MDD-based constraint programming. In *Principles and Practice of Constraint Programming (CP 2010)*, volume 6308 of *LNCS*, pages 266–280. Springer, 2010.

[95] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings, Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 279–288. Morgan Kaufmann, 1999.

[96] J. N. Hooker. *Integrated Methods for Optimization*. Springer, 2nd edition, 2012.

[97] J. N. Hooker. Decision diagrams and dynamic programming. In C. Gomes and M. Sellmann, editors, *CPAIOR Proceedings*, volume 7874 of *LNCS*, pages 94–110. Springer, 2013.

[98] K. Hosaka, Y. Takenaga, T. Kaneda, and S. Yajima. Size of ordered binary decision diagrams representing threshold functions. *Theoretical Computer Science*, 180:47–60, 1997.

[99] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, Department of Computer Science, 1995.

[100] A. Ignatiev, A. Morgado, V. Manquinho, I. Lynce, and J. Marques-Silva. Progression in maximum satisfiability. In *Proceedings, European Conference on Artificial Intelligence (ECAI)*, IOS Press Frontiers in Artificial Intelligence and Applications, 2014.

[101] ILOG. *CPLEX Optimization Studio V12.4 Manual*, 2012.

[102] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *Artificial Intelligence Magazine (AI Magazine)*, 1(33):89–94, 2012.

[103] C. Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik*, 70:185–190, 1869.

[104] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[105] M. Koshimura, H. Nabeshima, H. Fujita, and R. Hasegawa. Solving open job-shop scheduling problems by SAT encoding. *IEICE Transactions on Information and Systems*, 93:2316–2318, 2010.

[106] S. Kumar, A. R. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 763–773, 2012.

[107] V. Lagoon and P. J. Stuckey. Set domain propagation using ROBDDs. In M. Wallace, editor, *Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 347–361, 2004.

[108] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:959–975, 1994.

[109] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society B*, 50:157–224, 1988.

[110] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[111] M. Löbbing and I. Wegener. The number of knight's tours equals 13, 267, 364, 410, 532 – Counting with binary decision diagrams. *The Electronic Journal of Combinatorics*, 3, 1996.

[112] T. Lopes, A. A. Ciré, C. de Souza, and A. Moura. A hybrid model for a multiproduct pipeline planning and scheduling problem. *Constraints*, 15:151–189, 2010.

[113] M. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-based propagators for the SEQUENCE and related global constraints. In *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *LNCS*, pages 159–174. Springer, 2008.

[114] R. Martí, V. Campos, and E. Piñana. A branch and bound algorithm for the matrix bandwidth minimization. *European Journal of Operational Research*, 186(2):513–528, 2008.

[115] R. Martí, M. Laguna, F. Glover, and V. Campos. Reducing the bandwidth of a sparse matrix with tabu search. *European Journal of Operational Research*, 135(2):450–459, 2001.

[116] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings, Conference on Design Automation*, pages 272–277. IEEE, 1993.

[117] S.-I. Minato. πDD: A new decision diagram for efficient problem solving in permutation space. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *LNCS*, pages 90–104. Springer, 2011.

[118] T. Moisan, J. Gaudreault, and C.-G. Quimper. Parallel discrepancy-based search. In *Principles and Practice of Constraint Programming (CP 2013)*, volume 8124 of *LNCS*, pages 30–46. Springer, 2013.

[119] N. Narodytska. *Reformulation of Global Constraints*. PhD thesis, University of New South Wales, 2011.

[120] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[121] A. J. Orman and H. P. Williams. A survey of different integer programming formulations of the travelling salesman problem. In E. J. Kontoghiorghes and C. Gatu, editors, *Optimisation, Economics and Financial Analysis*, pages 933–106. Springer, 2006.

[122] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[123] G. Perez and J.-C. Régin. Improving GAC-4 for table and MDD constraints. In *Principles and Practice of Constraint Programming (CP 2014)*, pages 606–621, 2014.

[124] G. Perez and J.-C. Régin. Efficient operations on MDDs for building constraint programming models. In *Proceedings, International Joint Conference on Artificial Intelligence (IJCAI)*, pages 374–380, 2015.

[125] L. Perron. Search procedures and parallelism in constraint programming. In *Principles and Practice of Constraint Programming (CP 1999)*, volume 1713 of *LNCS*, pages 346–360. Springer, 1999.

[126] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.

[127] E. Piñana, I. Plana, V. Campos, and R. Martí. GRASP and path relinking for the matrix bandwidth minimization. *European Journal of Operational Research*, 153(1):200–210, 2004.

[128] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 3rd edition, 2008.

[129] W. B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley, 2nd edition, 2011.

[130] W. Pullan, F. Mascia, and M. Brunato. Cooperating local search for the maximum clique problem. *Journal of Heuristics*, 17(2):181–199, 2011.

[131] P. Refalo. Learning in search. In P. Van Hentenryck and M. Milano, editors, *Hybrid Optimization: The Ten Years of CPAIOR*, pages 337–356. Springer, 2011.

[132] J.-C. Régin. AC-*: A configurable, generic and adaptive arc consistency algorithm. In *Proceedings of CP*, volume 3709 of *LNCS*, pages 505–519. Springer, 2005.

[133] J.-C. Régin. Global constraints: A survey. In P. Van Hentenryck and M. Milano, editors, *Hybrid Optimization: The Ten Years of CPAIOR*, pages 63–134. Springer, 2011.

[134] J.-C. Régin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *Principles and Practice of Constraint Programming (CP 1997)*, volume 1330 of *LNCS*, pages 32–46. Springer, 1997.

[135] J.-C. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In *Principles and Practice of Constraint Programming (CP 2013*, volume 8124 of *LNCS*, pages 596–610, 2013.

[136] A. Rendl, M. Prandtstetter, G. Hiermann, J. Puchinger, and G. Raidl. Hybrid heuristics for multimodal homecare scheduling. In *CPAIOR Proceedings*, volume 7298 of *LNCS*, pages 339–355. Springer, 2012.

[137] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[138] S. Sanner and D. McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proceedings, International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1384–1390, 2005.

[139] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. Report on the experimental language, X10. Technical report, IBM Research, 2011.

[140] J. Saxe. Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM Journal on Algebraic and Discrete Methods*, 1:363–369, 1980.

[141] A. S. Schulz. The permutahedron of series-parallel posets. *Discrete Applied Mathematics*, 57(1):85 – 90, 1995.

[142] G. Shafer, P. P. Shenoy, and K. Mellouli. Propagating belief functions in qualitative Markov trees. *International Journal of Approximate Reasoning*, 1:349–400, 1987.

[143] G. Shani, P. Poupart, R. I. Brafman, and S. E. Shimony. Efficient ADD operations for point-based algorithms. In *Proceedings, International Conference on Automated Planning and Scheduling (ICAPS)*, pages 330–337, 2008.

[144] C. E. Shannon. A symbolic analysis of relay and switching circuits. Master's thesis, Massachusetts Institute of Technology, 1937.

[145] P. P. Shenoy and G. Shafer. Propagating belief functions with local computation. *IEEE Expert*, 1:43–52, 1986.

[146] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings of Conference on Neural Information Processing Systems*, pages 1089–1095, 2000.

[147] TSPLIB. Retrieved at `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/` on December 10, 2012, 2012.

[148] P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 85–134. Elsevier, 2006.

[149] P. Van Hentenryck and L. Michel. The Objective-CP optimization system. In *Principles and Practice of Constraint Programming (CP 2013)*, volume 8124 of *LNCS*, pages 8–29, 2013.

[150] P. Van Hentenryck and M. Milano, editors. *Hybrid Optimization: The Ten Years of CPAIOR*. Springer, 2011.

[151] W.-J. van Hoeve and I. Katriel. Global constraints. In P. Rossi, F. van Beek and T. Walsh, editors, *Handbook of Constraint Programming*, pages 205–244. Elsevier, 2006.

[152] W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. Revisiting the sequence constraint. In *Principles and Practice of Constraint Programming (CP 2006)*, volume 4204 of *LNCS*, pages 620–634. Springer, 2006.

[153] W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. New filtering algorithms for combinations of among constraints. *Constraints*, 14:273–292, 2009.

[154] P. Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In J.-C. Régin and M. Rueher, editors, *CPAIOR Proceedings*, volume 3011 of *LNCS*, pages 335–347. Springer, 2004.

[155] P. Vilím, P. Laborie, and P. Shaw. Failure-directed search for constraint-based scheduling. In *CPAIOR Proceedings*, volume 9075 of *LNCS*, pages 437–453. Springer, 2015.

[156] A. von Arnim, R. Schrader, and Y. Wang. The permutahedron of N-sparse posets. *Mathematical Programming*, 75(1):1–18, 1996.

[157] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, 2000.

[158] X10 programming language web site. http://x10-lang.org/, January 2010.

[159] Y. Zhao. The number of independent sets in a regular graph. *Combinatorics, Probability & Computing*, 19(2):315–320, 2010.

Index

0/1 programming, 15, 16

abstraction, 21
affine algebraic decision diagram, 20
algebraic decision diagram, 21
ALLDIFFERENT, 19, 76, 79
  filtering, 77, 78, 214
  MDD consistency, 175
AMONG, 176, 179, 183
  bounds consistency, 195
  MDD consistency, 176, 195
approximate dynamic programming, 21

bandwidth, 87
Bayesian network, 153
BDD, **26**
belief logic, 153
binary decision diagram, 13, **26**
  ordered, 14
  reduced ordered, 14
binary-decision program, 12
bound, 98
  from relaxed decision diagram, 19, 56
  from restricted decision diagram, 84
  versus maximum width, 66
bounds consistency, **162**, 161–162
  AMONG, 195
  linear inequality, 162
  SEQUENCE, 195
branch and bound, 3, 20, 138
  algorithm, **98**
  based on decision diagrams, 95–122
  MAX-2SAT, 108
  maximum cut problem, 104
  maximum independent set problem, 101
  parallel, 109–122
  variable ordering, 101
branch and cut, 15
branching, 3
  in relaxed decision diagram, 3, 20, 96

canonical reduced decision diagram, 27
canonical weighted decision diagram, 20, 139, 147, 148
clique cover, 102
compilation
  by separation, 46–50, 74–81
  top-down, 19, 30–32, 57–58, 85–86
conjoining, 50
consecutive ones property, 88
consistency
  bounds, *see* bounds consistency
  domain, *see* domain consistency
  MDD, *see* MDD consistency
constraint
  ALLDIFFERENT, 19, 76, 79, 175, 214
  AMONG, 176, 179, 183
  ELEMENT, 177
  equality, 173
  GEN-SEQUENCE, 194

© Springer International Publishing Switzerland 2016

D. Bergman et al., *Decision Diagrams for Optimization*, Artificial Intelligence: Foundations, Theory, and Algorithms, DOI 10.1007/978-3-319-42849-9

global, 3, 17, 19, 158
linear inequality, 161, 174
not-equal, 173
precedence, 207, 215, 218, 228
REGULAR, 169
SEQUENCE, 183–204
table, 18, 160, 168, 169
time window, 207, 215, 227
two-sided inequality, 174
UNARYRESOURCE, 161
constraint programming, 3, 17, 19, **158**, 157–164
constraint propagation, 158, 162
MDD-based, 164
search, 163
constraint propagation, 3, 18, 19, 158, 162–163
MDD-based, 166
constraint separation, 46
constraint-based modeling, 137
constraint-based scheduling, 206, 222–223
control variable, 28, 139
curse of dimensionality, 4, 137
cutset, *see* exact cutset
cutting plane, 16, 19

DD, *see* decision diagram
DDX10, 111, 116
decision diagram, 5, **25**
affine algebraic, 20
algebraic, 21
basic concepts, 25–27
binary, *see* binary decision diagram
canonical reduced, 27
canonical weighted, 139, 147, 148
exact, **26**
multivalued, *see* multivalued decision diagram
nonserial, 156
ordered, 14
polytope, 16
reduced, **27**
reduced ordered, 14
relaxed, 2, 6, 18, **55**, 55–81, 140
restricted, 3, **84**, 83–94
sound, 17
top-down compilation, 30–32
variable ordering, 15, 123–135
weighted, 20, 26, 146
width, 25
dependency graph, 154
domain consistency, **159**, 159–161
linear inequality, 161
UNARYRESOURCE, 161
domain filtering, 158
domain propagation, 18
domain store, 18, 164
DP, *see* dynamic programming
dynamic programming, 20, 28–29, 138, 139
approximate, 21
nonserial, 139, 153
stochastic, 21, 22
dynamic programming model
MAX-2SAT, 45–46, 52
maximum cut problem, 42–44, 50
maximum independent set problem, 33–34
proof of correctness, 50–54
set covering problem, 35–37
set packing problem, 37–39
single-machine scheduling, 40–42

ELEMENT, 177
equality constraint, 173
exact cutset, 97–98
frontier cutset, 100
last exact layer, 100
traditional branching, 100
exact decision diagram, **26**
compilation by separation, 46–50
top-down compilation, 30–32

feasibility pump, 84
Fibonacci number, 128
filtering, 18, **74**, 166, 173
ALLDIFFERENT, 77, 78, 214
makespan, 217
precedence constraint, 215
SEQUENCE, 190–196

single-machine scheduling, 77–78, 212, 214–218
sum of setup times, 217
time window constraint, 215
total tardiness, 218
frontier cutset, 100

GEN-SEQUENCE, 194
global constraint, 3, 17, 19, 158, *see also* constraint
graph coloring problem, 19

incremental refinement, 47, 57, 74, **75**
independent set, **32**
independent set problem, *see* maximum independent set problem
integer programming, 4, 46, 84, 86, 96, 141, 144
MAX-2SAT, 108
maximum cut problem, 105
maximum independent set problem, 32, 102
set covering, 34
set packing, 37
intersection of MDDs, 169–173
inventory management problem, 20, 151

k-tree, 153

last exact layer, 100
linear arrangement problem, 144
linear programming relaxation, 67
load balancing, 114
location theory, 153
logic circuit, 14
long arc, 27

makespan, 39, 76, 207, 209
market split problem, 18
Markov decision process, 21
Markovian, **29**
MAX-2SAT, 20, **44**
branch and bound, 108
dynamic programming model, 45–46, 52
integer programming model, 108
relaxed decision diagram, 63–64
maximal path decomposition, 128, 130
maximum 2-satisfiability problem, *see* MAX-2SAT
maximum cut problem, 20, **42**
branch and bound, 104
dynamic programming model, 42–44, 50
integer programming model, 105
relaxed decision diagram, 60–63
maximum flow problem, 16
maximum independent set problem, 19, 20, **32**, 123
bipartite graph, 126
branch and bound, 101
clique, 125
clique cover, 102
cycle, 126
dynamic programming model, 33–34
Fibonacci number, 128
integer programming model, 32, 102
interval graph, 127
linear programming relaxation, 67
parallel branch and bound, 116
path, 125
relaxed decision diagram, 59–60
tree, 127, 132
MCP, *see* maximum cut problem
MDD, **26**
MDD consistency, **167**, 167–169
ALLDIFFERENT, 175
AMONG, 176, 195
ELEMENT, 177
equality constraint, 173
linear inequality, 174
not-equal constraint, 173
SEQUENCE, 186–190, 195
table constraint, 168
two-sided inequality, 174
via MDD intersection, 171
MDD filtering, *see* filtering
MDD propagation, 166–167
via domain propagators, 178
via MDD intersection, 169–173

MDD store, 164, 166
merging heuristic, 64–65
  minimum longest path, 65
  minimum state size, 65
  random, 65
minimum bandwidth problem, 87, 144
MISP, *see* maximum independent set problem
mixed-integer programming, 3
modeling
  constraint-based, 137
  for decision diagrams, 4, 139
  in constraint programming, 4, 158
  in mixed-integer programming, 4
  recursive, 4, 138, 139
multivalued decision diagram, **26**

node refinement, 19
nonseparable cost function, 148
nonserial decision diagram, 156
nonserial dynamic programming, 139, 153
not-equal constraint, 173
nurse rostering, 181, 184, 201

ordered decision diagram, 14

parallel branch and bound, 109–122
  maximum independent set problem, 116
parallelization, 3, **109**
  centralized strategy, 112
  DDX10, 111, 116
  global pool, 113
  load balancing, 114
  local pool, 114
  SATX10, 116
  X10, 111
postoptimality analysis, 17
precedence constraint, 207, 215, 218, 228
primal heuristic, 3, 19, 83
product configuration, 15
propagation, *see* constraint propagation
propagation algorithm, 158

recursive modeling, 4, 138, 139
reduced decision diagram, 14, **27**
refinement, 19, **74**, 166
  incremental, 47
  single-machine scheduling, 79–81, 212, 219–220
REGULAR, 169
relaxed decision diagram, 2, 6, 18, **55**, 55–81, 98, 140
  bound, 56
  compilation by separation, 74–81
  incremental refinement, **75**
  MAX-2SAT, 63–64
  maximum cut problem, 60–63
  maximum independent set problem, 59–60
  merging heuristic, 64–65
  top-down compilation, 57–58
  variable ordering, 65
  width versus bound, 66
restricted decision diagram, 3, **84**, 83–94, 98
  bound, 84
  set covering problem, 86
  set packing problem, 86
  top-down compilation, 85–86
RO-BDD, *see* reduced ordered decision diagram
root state, 28
root value, 29

SATX10, 116
SCP, *see* set covering problem
search
  branch-and-bound, 3, 20, 96, 138
  branch-and-cut, 15
  branching, 3
  constraint programming, 163
separable cost function, 147
SEQUENCE, 183–204
  bounds consistency, 195
  filtering, 190–196
  MDD consistency, 186, 195
  width, 190
sequence-dependent setup times, 142, 207
sequencing, *see* single-machine scheduling
sequential ordering problem, 229
set covering problem, 19, 20, **34**, 147

dynamic programming model, 35–37
integer programming model, 34
restricted decision diagram, 86
set packing problem, 20, **37**
dynamic programming model, 37–39
integer programming model, 37
restricted decision diagram, 86
set variables, 18
single-machine scheduling, 39, 76–81, 141, 142, **207**
ALLDIFFERENT, 214
dynamic programming model, 40–42
filtering, 77–78, 212, 214–218
makespan, 39, 207, 209, 217
MDD representation, 208–210
precedence constraint, 207, 215, 218–219
refinement, 79–81, 212, 219–220
relaxed MDD, 211–214
sum of setup times, 207, 209, 217
time window constraint, 207, 215
total tardiness, 141, 207, 209, 218, 233–234
width, 79, 221
solution counting, 15
sound decision diagram, 17
SPP, *see* set packing problem
stable set problem, 19, 20
state space, 29, 139
MAX-2SAT, 45
maximum cut problem, 43
maximum independent set problem, 34
set covering problem, 35
set packing problem, 37
single-machine scheduling, 40
state space relaxation, 20
state transition graph, 27, 138, 140
state variable, 28
state-dependent cost, 20, 138, 146
state-graph, 31
stochastic dynamic programming, 21, 22
switching circuit, 12

table constraint, 18, 160, 168, 169
terminal state, 28
threshold decision diagram, 16
time window constraint, 207, 215, 227
top-down compilation, 19
exact decision diagram, 30–32
relaxed decision diagram, 57–58
restricted decision diagram, 85–86
transition cost function, 29
MAX-2SAT, 46
maximum cut problem, 43
maximum independent set problem, 34
set covering problem, 36
set packing problem, 38
single-machine scheduling, 41
transition function, 29, 139
MAX-2SAT, 45
maximum cut problem, 43
maximum independent set problem, 34
set covering problem, 35
set packing problem, 38
single-machine scheduling, 41
traveling salesman problem, 143
makespan objective, 230–232
precedence constraints, 228–229
time windows, 224, 227, 230–232

unary machine scheduling, *see* single-machine scheduling

variable ordering, 15, 123–135
in branch and bound, 101
k-look ahead, 131
maximal path decomposition, 130
maximum independent set problem
bipartite graph, 126
clique, 125
cycle, 126
Fibonacci number, 128
interval graph, 127
path, 125
tree, 127, 132
minimum number of states, 130
relaxation bound, 134
relaxed, 130–131
relaxed decision diagram, 65
width versus relaxation bound, 132

weighted decision diagram, 20, 26, 146
width, **25**, 124
  bound from decision diagram, 66
  maximum independent set problem
    bipartite graph, 126
    clique, 125
    cycle, 126
    Fibonacci number, 128
    interval graph, 127
    path, 125
    tree, 127
  SEQUENCE, 190
  single-machine scheduling, 79, 221

X10, 111