Carnegie Mellon University
Tepper School of Business

Doctoral Dissertation

Decision Diagrams for Optimization

Andre Augusto Cire

August, 2014

Submitted to the Tepper School of Business in Partial Fulfillment of the Requirements for the Degree of Doctor in Operations Research

Dissertation Committee:
John N. Hooker (co-chair)
Willem-Jan van Hoeve (co-chair)
Michael Trick
Meinolf Sellmann

Abstract

Decision diagrams are compact graphical representations of Boolean functions originally introduced for applications in circuit design, simulation, and formal verification. Recently, they have been considered for a variety of purposes in optimization and operations research. These include facet enumeration in integer programming, maximum flow computation in large-scale networks, solution counting in combinatorics, and learning in genetic programming techniques.

In this dissertation we develop new methodologies based on decision diagrams to tackle discrete optimization problems. A decision diagram is viewed here as a graphical representation of the feasible solution set of a discrete problem. Since such diagrams may grow exponentially large in general, we work with the concept of approximate decision diagrams, first introduced by Andersen et al (2007). An approximate decision diagram is a graph of parameterized size that represents instead an over-approximation or under-approximation of the feasible solution set. Thus, it can be used to obtain either bounds on the optimal solution value or primal solutions to the problem.

As our first contribution, we provide a modeling framework based on dynamic programming that can be used to specify how to build a decision diagram of a discrete optimization problem and how to approximate it, which facilitates the encoding process of a problem to a diagram representation. We then present a branching scheme that exploits the recursive structure of an approximate diagram, establishing a novel generic solver for discrete optimization problems. Computational results in classical optimization problems show that more instances can be solved in less computation time using our approach than mathematical programming techniques. In particular, we were able to reduce the known optimality gap of benchmark instances of the maximum cut problem.

In our second contribution, we focus on the application of approximate diagrams to particular domains; namely, to sequencing problems, common in the context of routing and scheduling, and to timetable problems. We indicate that, besides the computation of bounds, approximate decision diagram can be used to deduce non-trivial constraints of a problem, such as precedence relations between jobs in scheduling applications. We show that such inference can be incorporated into state-of-the-art solvers and speed-up the optimization process by orders of magnitude.

Finally, we propose new parallelization strategies that exploits the recursive structure of an approximate diagram. These strategies decouple a problem in a naturally loose fashion and allow for more effective load balancing heuristics when considering hundreds of computer cores.

Acknowledgements

First and foremost, I would like to express my appreciation and deepest gratitude to my advisors, John Hooker and Willem-Jan van Hoeve. They are my role models as researchers and provided me invaluable advice, encouragement, and challenging and exciting problems to work on throughout the program. More than that, they represent what I hope to become as a person, and I feel honored and privileged to be advised by them. I am also indebted to Meinolf Sellmann, a person with an exceptional intelligence and perception whom I have a profound admiration for, and Michael Trick, for being part of my dissertation committee and for his thoughtful comments.

This work would have not been possible without the patience and support of a number of esteemed friends and colleagues, including Marco Molinaro, Alex Kazachkov, Ishani Aggarwal, Elvin Coban, Amitabh Basu, Sam Hoda, Horst Samulowitz, Ashish Sabharwal, Qihang Lin, Andrea Qualizza, Amin Sayedi, Nandana Sengupta, Vince Slaugh, Jessie Wang, Abha Kapoor, Hellen, Thiago Serra, and Christian Tjandraatmadja. Moreover, I would like to thank Lawrence Rapp, the best Ph.D. assistant director I could hope for, and all the Tepper School of Business staff, who provided me the support I needed to complete the program.

I owe a special thanks to my colleague and great friend David Bergman. David has an extraordinary intelligence and an admirable personality that are rarely found in any person. I greatly appreciated the time we spent on our adventures, either travelling nonstop to conferences, or during our sleepless working sessions. The majority of the results in this dissertation was done in collaboration with David. It has been an immense pleasure, and I hope we can continue producing much more in the future.

I also would like to thank Selvaprabu Nadarajah and Negar Soheili, two of the best friends a person can have. I have no words to express how important was their support, encouragement, and company throughout these years. I learnt a lot from all the time we spent together. Their friendship is invaluable to me, and I am forever indebted to them.

Last but not least, I want to thank my parents, Antonio and Maria Antonia, and my sister, Danielle, for their unconditional love and patience with me. They constantly supported me throughout my life and have always been pivotal in anything I have accomplished so far. I dedicate this dissertation to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimization has recently stood out as an essential tool for the quantitative and analytical solutions of modern business problems. Applications that involve optimization are virtually ubiquitous in our society: They are used to define how crews are scheduled in the flights we take, how ads are displayed in the web pages we visit, how our mailed packages are routed to reach their destinations, how banks manage our investments, and even the order of songs in online radios. Fueled by the increasing availability of data and computer resources, the range of applications tends to grow even more pronouncedly in the next few years.

One major factor in this trend is the remarkable advancement of optimization techniques in recent years. In particular, general-purpose optimization methods, such as mathematical programming and constraint programming, have played a key role in this context. They provide a language from which a user can model their problems, and apply sophisticated mathematical techniques to propose high-quality solutions to such models. Optimization techniques have improved significantly in the last decades, as reflected by the substantial speed-ups in solving time. Problems that would take hours to solve in the past now take just a few seconds, primarily due to algorithmic advancements, not only computer power [125]. Moreover, optimization techniques are also integrated into other decision-making tools, such as spreadsheets or business analytics software, making them more accessible and facilitating their use.

However, there are still a wide range of problems that remain extremely challenging for state-of-the-art generic optimization solvers. The reasons for this vary greatly. For example, they may contain some combinatorial structure that is hard to exploit in existing techniques, such as in the case of the classic maximum-cut problem [118]. Or they may involve a huge amount of data that would result in models with thousands or millions of variables and constraints, as found in inventory routing problems [38], which cannot be handled directly by any available optimization software up to this date. These two reasons alone are sufficient to motivate the research on new generic optimization technologies that show potential to tackle these issues, ultimately extending the libraries of problems that can be more easily modeled and solved.

With this project in mind, the work in this dissertation presents novel generic solving methodologies based on *decision diagrams*, which have recently brought a new perspective to the field of optimization. A decision diagram is a graphical data structure originally introduced for representing Boolean functions [98, 3], with successful applications in circuit design and formal verification [32, 89]. Their introduction in optimization is recent and includes facet enumeration in integer programming, maximum flow computation in large-scale networks, solution counting in combinatorics, and learning in genetic programming [16, 132]. As we will show in this dissertation, we expand this line of research by introducing new ways of formulating and solving optimization problems with decision diagrams. We also study their structural properties and indicate ways on how these techniques can be used in conjunction with other methods, with a particular focus on mathematical programming and constraint programming.

To achieve this, the work throughout this dissertation relies on the decision diagram framework established in previous studies [16, 132]. The fundamental concept is to use decision diagrams as a graphical structure to compactly represent a set of solutions to a problem. For example, consider a typical optimization problem, particularly formulated as an integer linear program:

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

A decision diagram for the problem above represents possible assignments of the variables $x_1, \ldots, x_5$ and is depicted in Figure 1.1. It is a directed acyclic graph where all paths start at a root node r and end at a terminal node t. The nodes are partitioned into 6 layers so that an arc leaving a node at layer i corresponds to a value assignment for variable x_i. In particular, since all variables are binaries for this problem, there are two types of arcs: dashed arcs at layer i lead to the assignment $x_i = 0$ and solid arcs lead to the assignment $x_i = 1$. Hence, any path from r to t represents a complete value assignment for variables x. One can verify that the diagram in Figure 1.1 exactly represents the 7 feasible solutions of problem (1.1), each compactly represented by one path from the root node r to the terminal node t in the diagram.

Also, with each arc we associate a *weight* which evaluates to the contribution of that value assignment to the objective function. Dashed arcs have thus a weight of zero, 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

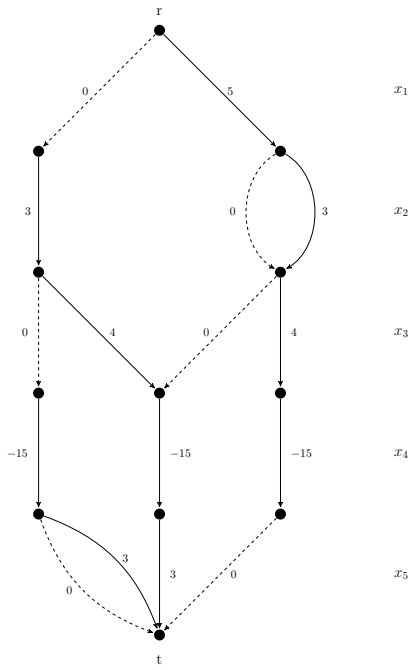


Figure 1.1: Example of a decision diagram for problem (1.1). Arcs are partitioned into layers, one for each problem variable. Dashed and solid arcs at layer i represent the assignments $x_i = 0$ and $x_i = 1$, respectively.

with respect to these arc weights. For the diagram in Figure 1.1, the longest path has value -3 and indicates the assignment $x_1 = x_2 = x_3 = x_4 = 1$ and $x_5 = 0$, which is the optimal solution of the integer programming problem (1.1). Analogously, any linear function (or, more generally, any separable function) can be optimized in polynomial time in the size of the diagram. This optimization property, alongside the potential to compactly represent a set of solutions to a problem, are the main incentives for the majority of works that exploited their use in the field of optimization so far [16, 132].

Nonetheless, a clear issue with this framework is that a decision diagram representing exactly the feasible solutions to an optimization problem can grow exponentially large in the number of variables of the problem. Indeed, since optimizing a linear function over a decision diagram is equivalent to a minimum-cost flow computation from the root r to the terminal node t, theory from extended formulations in integer programming demonstrates that all decision diagrams for some problem instances will undoubtedly have exponential size. This is the case, for example, of certain independent set problem instances [56]. Most of practical computations using exact decision diagrams are thus prohibitive in general, as only problems with very few variables can be handled that way.

To circumvent this issue, the authors in [5] introduced the concept of a *relaxed decision diagram*, which is a diagram of limited size that represents instead an over-approximation of the solution

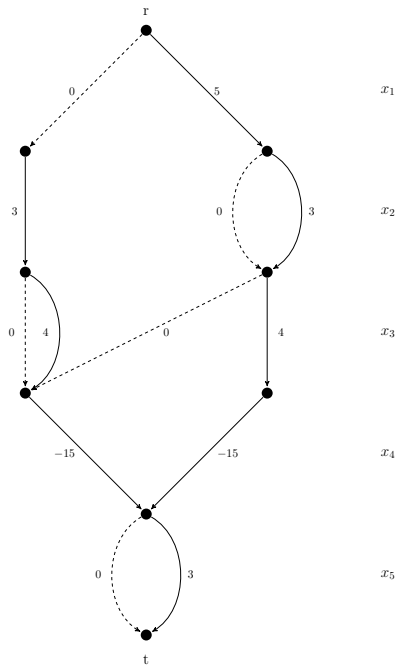


Figure 1.2: Example of a relaxed decision diagram for problem (1.1).

set of an optimization problem. 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 allowed *width*, i.e. the maximum number of nodes in any layer. A major property of relaxed diagrams is that, since they represent an over-approximation of the solution set, a longest path now yields an upper bound on the maximum value of an objective function (and equivalently, a shortest path yields a lower bound for minimization problems).

For example, Figure 1.2 depicts a relaxed decision diagram for problem (1.1) with a limited width of 2. Even though one can observe that all feasible solutions of (1.1) are represented by some path in this relaxed decision diagram, not all paths correspond to feasible assignments. In particular, the longest path represents the assignment $x_1 = \cdots = x_5 = 1$ and has a value of 0, which is an upper bound of the optimal solution value, -3. Such dual bound can be potentially better than the ones provided by other generic technologies. For example, the typical linear programming relaxation for problem (1.1), obtained by replacing the integrality constraints by $0 \leq x_1, \ldots, x_5 \leq 1$, yields a relatively worse upper bound of 5.25.

Relaxed decision diagrams were initially proposed in [5] as an alternative to the domain store relaxation that is commonly used in constraint programming solvers. The authors demonstrated that relaxed diagrams may reveal inconsistent variable assignments that other techniques are unable to, property which is crucial to the constraint programming paradigm. For instance, we can deduce from the relaxed decision diagram in Figure 1.2 that $x_4 = 1$ in any feasible solution. The same deduction is not possible to obtain, for example, from straightforward domain consistency of the

linear system. Following the original work from [5], the authors in [78] and [85] developed generic methods for systematically compiling relaxed diagrams for constraint programming models. The first work to consider relaxed decision diagrams for the purpose of obtaining optimization bounds is from [26], where lower bounds for particular set covering instances were compared to the ones provided by integer programming technology. In this dissertation we extend this line of research by further exploiting the structural properties of decision diagrams, studying both modeling and solving aspects for their use as generic tools in optimization.

1.1 Contributions and Outline

The main goal of this work is to improve the solving capabilities of generic optimization technology through the use of decision diagrams. After showing the previous works that led to this research in Chapter 2, the goal is achieved here by our five main contributions, described below. All of these contributions focus on *discrete optimization problems*, which are naturally suitable to a decision diagram representation.

1. In Chapter 3 we propose a modeling framework based on dynamic programming that can be used to specify how to build decision diagrams for an optimization problem. The objective is to demonstrate that decision diagrams permit a more flexible approach to modeling, since no particular structure is imposed on the constraints or objective function of the problem. Instead, it only requires the problem to be formulated in a recursive way. This may be more appropriate for problems with no adequate integer linear formulation, such as the maximum cut problem. Furthermore, the framework can also be beneficial for problems where a recursive formulation is much more compact than an explicit description the constraints; consider, for example, a maximum independent set problem with millions of vertices and edges.

   The relationship between decision diagrams and dynamic programming is often mentioned in previous studies, such as in [16], and it is exploited thoroughly in [87]. The work presented here extends those ideas and formalizes the construction of decision diagram from a dynamic programming model. We provide several examples of classical optimization problems modeled in this framework.

2. Optimization bounds are essential for many solving techniques in optimization, specially for those that rely on branch-and-bound strategies. In Chapter 4, we enhance our modeling framework in order to provide a generic tool for the compilation of relaxed decision diagrams. We then present an analysis of the different parameters that influence the quality of the optimization bound it provides, such as how the problem variables are associated with each of the diagram layers (or *variable ordering*). Using the maximum independent set problem as a test case and a suitable choice of such parameters, we find that a decision diagram can

deliver tighter optimization bounds than those obtained by a strong linear programming (LP) formulation of the problem. This holds even when the LP is augmented by cutting planes generated at the root node by a state-of-the-art integer programming solver with non-default parameters as well. These optimization bounds are also obtained in less computation time for most of the traditional benchmark instances of the maximum independent set problem.

3. In Chapter 5, we introduce a new type of limited-size diagram in optimization, the *restricted decision diagrams*. They define instead an under-approximation of the solution set of an optimization problem. That is, any path from the root node to the terminal node in a restricted decision diagram corresponds to a a feasible solution of the optimization problem at hand, but not all feasible solutions are represented by a path in the diagram. Thus, a restricted decision diagram can be used as a general-purpose primal heuristic, similar to the *feasibility pump* technique in integer programming [58]. We show that the quality of the solutions provided by restricted decision diagram can also be superior to integer programming technology for structured set packing and set covering instances, specially for problems involving thousands of variables and constraints.

4. In Chapter 6, we introduce a branch-and-bound method based solely on relaxed and restricted decision diagrams. It differs considerably from traditional enumerative methods, such as those used in mathematical programming and constraint programming. The key idea is to branch on the *nodes* of either a relaxed or restricted diagram, which eliminates symmetry in search since each diagram nodes may potentially aggregate several equivalent partial solutions. We provide experiments comparing our branching techniques against state-of-the-art generic optimization technology for different problem classes. In particular, we are able to reduce the optimality gap of benchmark maximum cut instances that still remain unsolved.

   In addition, we propose the use of a decision diagrams in *parallel optimization* by turning the branch-and-bound procedure into a distributed process. Parallel optimization techniques have proved to be a challenge in practice, as current state-of-the-art integer programming and constraint programming solvers are not able to exploit more than a couple of dozen computer cores. We aim to show that, on the other hand, our techniques may be able to exploit the recursive structure of a decision diagram to yield parallel strategies suitable to massive computers with hundreds and thousands of cores. Moreover, relaxed decision diagrams can also be perceived as an approximation of a branching tree, providing measures that can be used in load balancing heuristics, for example.

5. We present two practical applications of relaxed decision diagrams in the area of scheduling. The first application is developed in Chapter 7 and concerns *sequencing problems*, which are those where the best order for performing a set of tasks must be determined. They are prevalent in manufacturing and routing applications, including production plants where jobs

should be processed one at a time in an assembly line, and in mail services where packages must be scheduled for delivery on a vehicle. The second application, presented in Chapter 8, refers to *timetabling problems*, in which we need to assign a resource to discrete points in time during a given horizon. For example, resources can be seen as employee shifts or rest days, and time points may represent the days of the week.

One of our key contributions from these applications is to demonstrate the *inference* power of a decision diagrams, i.e. how effective are the new problem constraints deduced from a relaxed decision diagram. We show that, besides straightforward domain restrictions (such as $x_4 = 1$ from our example in Figure 1.2), we can infer highly structured constraints of a problem from a relaxed decision diagram, such as non-trivial precedence relations that must hold between tasks in a scheduling application. This inference provides a method to link decision diagrams with other optimization techniques, since the deduced constraints may be applied, e.g., to strengthen an integer programming formulation of the problem or to derive new filtering algorithms in constraint programming.

In the studied applications, we show that this inference may be used to speed up constraint-based schedulers by orders of magnitude. The technique was particularly used to solve some open benchmark instances of the traveling salesman problem with precedence constraints, which still pose a challenge to current solvers.

Final remarks are then discussed in Chapter 9.

The work in this dissertation is based on a collection of papers published during the author's doctoral program. The papers were written in collaboration with a number of authors, which were invaluable and crucial to the development of this work. In particular, Chapter 3 and 6 are based on [23, 21, 42, 24], and on an unpublished paper under review entitled *Discrete Optimization with Decision Diagrams* by David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John Hooker. Chapter 4 is based on [21, 41], Chapter 5 is based on [25], Chapter 7 is based on [41], and Chapter 8 is based on [22].

Chapter 2

Related Work

The work on decision diagrams spans more than 30 years of research with dozens of papers concerning a most diverse range of applications. They were first introduced by Lee [98] for the purpose of representing *switching circuits*. A switching circuit is defined by a network of "ideal" switches; that is, each switch has exactly two exclusive states, such as *on/off* or *open/close*, and can be applied to model, for example, computer or telecommunications systems. Switching circuits might also include *memory*, in the sense that the state of a certain switch might be dependent on the state of past switches in the network. For example, Figure 2.1 extracted from [98] presents a simple switching circuit. The circuit has three switches, or *variables*, x, y, and z. Each variable can assume a value of 0 ("off") or 1 ("on"); when $x = 0$, we follow the path represented by x' in the figure, and if $x = 1$ we follow the path through x; analogously to the other variables. An output of 0 is generated when there is no value to follow, and 1 if we reach the end of the circuit. For instance, the switches $x = 1$, $y = 1$, and $z = 1$ leads to an output of 1 in Figure 2.1, and $x = 0$, $y = 0$ to an output of 0.

The seminal work on switching circuit theory was provided by the Master's thesis of Claude E. Shannon [123], a pioneer researcher in computer science. Shannon demonstrated that switching circuits are representable by symbolic logic, and thus could be written and manipulated in a language that followed the rules of Boolean algebra. Lee [98] wanted to provide an alternative representation to Shannon's algebraic language in order to facilitate the actual computation of the outputs given by switching circuits. To this end, Lee introduced the concept of *binary-decision program*. A binary-decision program was a computer program based on a single instruction T,

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

which states that, if a variable x assumes a value of 0, go to the instruction in address A; if otherwise x is 1, go to the instruction in address B. As mentioned in [98], representing switching circuits as a program facilitates computation because it defines a sequential description of the possible events

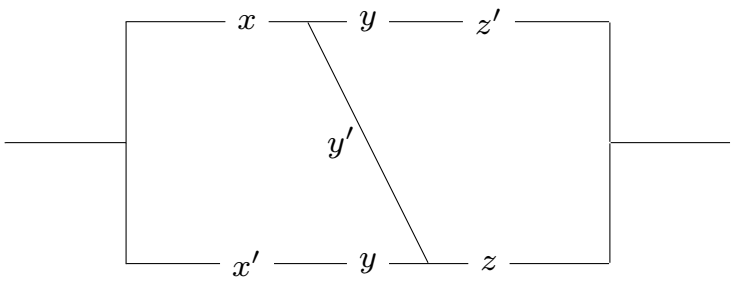


Figure 2.1: Example of a switching circuit from [98].

that may occur. For example, let $\mathcal{F}$ represents an output of 0, and $\mathcal{T}$ an output 1. Then, as described in [98] the circuit in Figure 2.1 can be represented by the binary-decision program

1. x; $2, 4$
2. y; $\mathcal{F}, 3$
3. z; $\mathcal{F}, \mathcal{T}$
4. y; $3, 5$
5. z; $\mathcal{T}, \mathcal{F}$

In [98], Lee established rules to construct binary-decision programs from the logical requirements of a switching circuit. He also provided upper and lower bounds on the minimum number of instructions that were necessary to construct a switching circuit. In particular, Lee formally showed that computing circuits through binary-decision programs was in general faster than computing them through *and*/*or*/*sum* operations from Boolean programs, usually by orders of magnitude.

The next landmark in the area was achieved by Akers [3]. Akers is the first to introduce a graphical structure, denoted by *binary decision diagram* (BDD), to represent switching networks encoded in functional form, the *switching functions*. A binary decision diagram is an acyclic directed graph where nodes represent variables and the arcs leaving a node represent value assignments for that variable. There are two *terminal* nodes, representing the outputs 0 and 1. Figure 2.2 depicts an example of a binary decision for the switching (or Boolean) function $f = (x \textbf{ xor } y \textbf{ xor } z)$. Notice that the paths in the BDD encode all *true* (output 1) and *false* (output 0) assignments of the variables x, y, z.

In practice, binary-decision programs and binary decision diagrams can be regarded as equivalent representations. The advantage of using graphical structures is that they are easier to manipulate and provide an *implementation-free* description of a Boolean function. More specifically, they can be used to derive different implementations aimed at computing outputs. One must note the dual role that the use of a graphical structure ensues: a BDD can be regarded both as a *representation* for Boolean functions as well as a *computational model*. For instance, in [3], Arkes applies

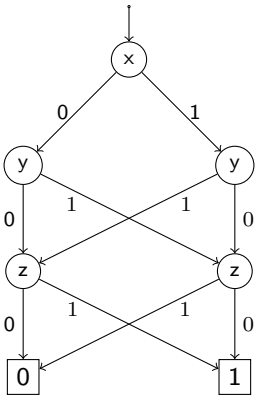


Figure 2.2: Example of a binary decision diagram for the switching function $f = (x \textbf{ xor } y \textbf{ xor } z)$.

BDDs to analyze certain types of Boolean functions, which relates to representation issues, and also as a tool for *test generation*, i.e. finding a set of inputs which can be used to confirm that a given implementation performs correctly, which exploits the computational model provided by a BDD.

However, the seminal work in the field was developed by Bryant [32]. Bryant also applied BDDs for the purpose of representing Boolean functions, but his data structure had a key difference when compared to the works of Lee and Arkes. Namely, the decision variables in his BDD representation were restricted to a particular *ordering*, forcing all nodes in a "layer" of the diagram to be related with the same decision variable (such as in Figure 2.2). Using this property, Bryant was able to describe BDDs in a *canonical form*; i.e. every Boolean function has a unique BDD representation for a fixed variable ordering. Such uniqueness is obtained by means of a *reduction* operation: Given any BDD for a Boolean function, the unique BDD representation is attained by superimposing nodes with isomorphic subgraphs. The unique BDD with the variable ordering restriction was then denoted by *Reduced Ordered Binary Decision Diagram* (RO-BDD).

As presented in [32], an important advantage of RO-BDDs is that several Boolean function manipulations could be performed efficiently over the diagram, such as complementing a function or combining two or more functions (e.g., through the use of **and**/**or** operators). In particular, Bryant showed that the time complexity for any single operation is bounded by the product of the BDD sizes representing the functions being operated on. However, he also noted in [32] that the variable ordering must be fixed in the input and can have a significant impact on the final size of the BDD. Computing the variable ordering that yields the smallest BDD is a coNP-Complete problem [63], thus heuristics that take into account the problem domain may be crucial in obtaining small BDDs for practical applications.

The canonical representation and the efficient operation algorithms provided by Bryant in [32] were key factors for a large trend of research in BDDs in computer science. Several variants of the basic BDD data structure model were proposed for different theoretical and practical purposes; we

refer to the monograph by Wegener [132] for a list of different BDD types, theoretical aspects, and their uses in practice. Relevant applications of BDDs include formal verification (also check [89]), model checking, CAD applications, product configuration, and genetic programming.

Decision Diagrams in Optimization

One of the early applications of decision diagrams that relates to optimization is on *solution counting* for general combinatorial problems. The idea is to perceive the constraints of a problem as a Boolean function $f(x)$ representing whether a solution x is feasible (output 1) or not (output 0). One can then create BDDs representing exactly the set of feasible solutions to a problem by omitting the paths leading to the output 0, and then counting the number of solutions encoded by a BDD using traditional graph algorithms. In general, solution counting through BDDs is usually ineffective as the diagram tends to grow too quickly on the number of variables. However, techniques combining BDDs, backtracking, and divide-and-conquer can be used to solve classical combinatorial chess problems efficiently, such as counting the number of knight's tours [99].

Using the same representation concept, Lai et al [97] proposed one of the initial uses of BDDs focusing exclusively on optimization problems. The paper presents a compilation method to encode binary integer programs (binary IPs) as Boolean functions. It also proposes a branch-and-bound algorithm that employs such BDDs to speed up search. The idea is to start solving the problem using traditional IP techniques (i.e., methods based on linear programming relaxations enhanced with cuts). Then, after some branching rounds, a BDD representing all feasible solutions to the corresponding small subproblem is compiled, and the corresponding optimal solution is extracted so that no more branching is necessary. Computational experiments were limited to a small number of instances, but showed a significant improvement over traditional IP methods. We remark in passing that [132] also presents other ways to formulate binary integer linear programs as BDDs (e.g., exploiting the use of affine functions), which were never tested experimentally.

BDDs have also been applied for maximum flow computation in large-scale 0-1 networks by Hacthel and Somenzi [73]. The authors developed an implicit network flow algorithm where BDDs were used to enumerate *flow augmenting paths*. Starting with a flow of 0, the corresponding flow augmenting BDD was compiled and analyzed to compute the next flow, and the process was repeated until the maximum flow was found (i.e., no more augmenting paths were found, giving an "empty" BDD). Hacthel and Somenzi were able to compute maximum flow for graphs having more than 10^{27} vertices and 10^{36} edges. However, this is only possible for graphs having short augmenting paths, since otherwise the resulting augmenting path BDDs would be too large.

A more recent and key work in the use of BDDs for optimization is the PhD dissertation of Behle [16]. Behle studied the relationship between a BDD representing the feasible solutions to a 0/1 IP model, which he called a *Threshold BDD*, and the polyhedral structure of the problem. Among his main results, Behle showed how BDDs could be used to enumerate vertices or facets of

the corresponding 0/1 polytope, and provided an IP model for finding the variable ordering that yields the BDD with the minimum size for a given problem. He also presented a new technique to generate valid inequalities for 0/1 IPs, which was effective for small but hard combinatorial optimization problems.

Again in the context of IPs, Hadzic and Hooker [77] applied BDDs to characterize the set of optimal or near-optimal solution to a general integer programming problem. Such BDDs could then be used to perform postoptimality or sensitivity analysis. One example is on *cost-based domain analysis*: Given a BDD enumerating the set of optimal or near-optimal solutions of a problem, this analysis calculates how the domain of a variable (i.e., the values the variable can assume) grows as one permits the cost to deviate further from optimality. Thus, a given variable may take only one value in any optimal solution, but as one considers solutions whose cost is within 1%, 2% or 3% of optimality, additional values become possible. This type of analysis can tell a practitioner that there is little choice as to the value of certain variables if one wants a good solution. The authors illustrated the analysis on capital budgeting and on network reliability problems.

A follow-up paper to the previous work was written by the same authors in [75]. The paper addresses the computational issue of how to minimize the growth of the BDD as the problem size increases. To this end, the authors examine the strategy of representing only near-optimal solutions, since these are generally the solutions of greatest interest to practitioners. This was done through the operations of *pruning*, where arcs identifying solutions below a certain objective function value were removed, and *contraction*, where nodes of the BDD were merged so as to reduce its size while adding solutions with a bounded cost to the BDD representation. A number of computational experiments were carried out over randomly generated 0/1 programs and showed promising results, as the near-optimal BDDs were relatively small compared to the optimal BDDs.

In the same year of [75], Andersen et al [5] published the work that plays a fundamental role in the concepts presented in this dissertation. The authors proposed the use of limited-size decision diagrams as a *discrete relaxation* to arbitrary constraint satisfaction problems. That is, decision diagrams would be applied to represent a superset of the feasible solution space of a discrete optimization problem, instead of encoding the feasible solutions exactly as in previous works. The size of such *relaxed* decision diagrams was controlled by a parameter given as input; the larger the size, the closer the relaxed decision diagram would be from the exact problem representation. In particular, the authors considered *multivalued decision diagrams* (MDDs), which are diagrams where variables can have arbitrary integer domains.

Relaxed MDDs in [5] were primarily applied as an alternative *constraint store* for constraint programming (CP). Namely, the key idea in the CP paradigm is to provide a clear separation between *model* and the *algorithms* to solve it [86]. Models are formulated by constraints representing rich substructures of a problem, such as a network flow or a knapsack constraint. Each of these constraints is associated with an algorithm that performs *inference*, i.e. that exploits the sub-

structure in order to deduce new constraints to strengthen a particular relaxation of the problem. Once a sufficient number of constraints has been added to the relaxation, its optimal solution will be feasible to the original problem (perhaps after some branching as well). In CP, the inferred constraints are collected in a so called *constraint store*, which effectively represents the relaxation of the problem. Traditional CP techniques use a *domain store* to this purpose, defined by the Cartesian product of the variable domains. Thus, the inferred constraints take the form of variable domain reductions, and a solution is found once all domains are singletons.

Andersen et al demonstrated in [5] that the domain store could be potentially ineffective to capture global information of the problem. The authors then proposed the use of relaxed MDDs as a new constraint store, where inference was represented by removal of arcs and addition of nodes in accordance to each constraint substructure, such as in the case of equality constraint demonstrated in [74]. Experimental results on constraint programming models composed of multiple structured constraints indicated that the relaxed MDD could speed up solving times by orders of magnitude.

Following that work, Andersen et al [78] and Hoda et al [85] developed generic methods for systematically compiling relaxed MDDs for CP models, as we will describe in different sections of this dissertation. The fundamental idea of their approaches is to construct the diagram in an incremental fashion, associating a particular *state* information with the MDD nodes so as to indicate how new nodes and arcs should be added to the diagram. Computational results provided by the authors in both papers also show orders of magnitude speed ups for a variety of CP problems.

The use of a node state information was also exploited by [19] for the purpose of obtaining bounds for set covering problems. The computation study provided by the authors indicated that the resulting bounds were superior to the ones obtained by a continuous relaxation of the problem, specially for instances having a constraint matrix with small bandwidth. The authors also propose a *top-down compilation* method to construct relaxed MDDs, which is the basis of the compilation method discussed in this dissertation.

In particular, our compilation method is based on a dynamic programming (DP) formulation of a discrete optimization problem. The relationship between MDDs and DPs was studied by Hooker in [87]. The author interprets an MDD as a representation of the DP state transition graph and incorporates state-dependent costs in the theory of MDDs. Furthermore, the paper shows that, for a given optimization problem and variable ordering, there is an unique MDD with "canonical" edge costs. An illustration of this analysis on a standard inventory management problem shows that the DP state transition graph can be greatly simplified when represented by its corresponding canonical MDD.

In a related note, relaxed MDDs can also be associated with state-space relaxations [39] or with DP consistency methods for constraint programming, as in [124]. However, these relationships still remain largely unexplored and provide the basis for future research.

Chapter 3

Exact Decision Diagrams

3.1 Introduction

In this chapter we introduce a modeling framework based on dynamic programming 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* and a compilation method based on *constraint separation*. We also present a study on how the structural properties of a combinatorial problem impacts the size of the exact decision diagram that represents its solution space. To this end, we focus on the independent set problem, taking advantage of the graph structure associated with the problem instance.

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 dissertation. Section 3.3 presents the modeling framework and the top-down compilation procedure, which are then exemplified in a number of classical optimization problems. Section 3.4 presents an alternative approach to compilation, *construction by separation*, where classes of the problem constraints are considered one at a time. Finally, in Section 3.5 we analyze bounds on the size of an exact decision diagram for the independent set problem.

3.2 Concepts and Notation

In this work we will focus on *discrete optimization problems*. For our purposes, a discrete optimization problem $\mathcal{P}$ has the form

$$\begin{aligned} \max \;\; & f(x) \\ & C_i, \;\; i = 1, \dots, m \\ & x_j \in D_j, \;\; j = 1, \dots, n \end{aligned} \tag{$\mathcal{P}$}$$

where $x = (x_1, \ldots, x_n)$ is a tuple representing the decision variables, $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a (possibly empty) constraint set, and each variable x_j has a finite domain D_j. Let $D = D_1 \times \cdots \times D_m$ be the Cartesian product of the domains. A constraint C_i states an arbitrary relation between two or more variables; it is *satisfied* by a variable assignment $\hat{x} \in D$ if the relation is observed by $\hat{x}$, and it is *violated* otherwise. Also, $f : D \to \mathbb{R}$ is the *objective function* of $\mathcal{P}$. A *feasible solution* of $\mathcal{P}$ is any $\hat{x} \in D$ that satisfies all of the constraints in $\mathcal{C}$. The set of feasible solutions of $\mathcal{P}$ is denoted by $\mathrm{Sol}(\mathcal{P})$. A feasible solution x^* is *optimal* for $\mathcal{P}$ if it satisfies $f(x^*) \geq f(x)$ for all $x \in \mathrm{Sol}(\mathcal{P})$. Finally, we let $z^* = f(x^*)$ be the optimal value.

EXAMPLE 1 A classical example of discrete optimization problem is the *0/1 knapsack problem*. Given items $1, \ldots, n$, each associated with a weight w_j and a value v_j, we wish to select items so as to maximize the resulting total value while keeping the weight capacity between a lower bound L and an upper bound U. The knapsack problem can be formulated as the following discrete optimization problem.

$$\begin{aligned} \max \ & \sum_{j=1}^{n} v_j x_j \\ & L \leq \sum_{j=1}^{n} w_j x_j \leq U \\ & x_j \in \{0,1\}, \ \ j = 1, \ldots, n \end{aligned} \tag{3.1}$$

In the formulation above, we define a variable x_j for each item j with binary domain $D_j = \{0, 1\}$, indicating whether item j is selected ($x_j = 1$) or not ($x_j = 0$). The objective function is the total value of the selected items, $f(x) = \sum_{j=1}^{n} v_j x_j$, and the constraint set is composed of a single constraint that maintains the weight capacity within L and U, i.e. $\mathcal{C} = \{L \leq \sum_{j=1}^{n} w_j x_j \leq U\}$.

Consider now the following 0/1 knapsack problem with four variables:

$$\begin{aligned} \max \ & x_1 + 12x_2 + 3x_3 + 4x_4 \\ & 5 \leq 5x_1 + 7x_2 + 2x_3 + 3x_4 \leq 8 \\ & x_j \in \{0,1\}, \ \ j = 1, \ldots, 4 \end{aligned} \tag{3.2}$$

The optimal solution to the problem $\mathcal{P}$ above is $x^* = (0, 1, 0, 0)$ and the optimal value is $z^* = 12$. The set of feasible solutions is $\mathrm{Sol}(\mathcal{P}) = \{(0, 0, 1, 1), (0, 1, 0, 0), (1, 0, 1, 0), (1, 0, 0, 1), (1, 0, 0, 0)\}$. □

A *decision diagram* (DD) $B = (U, A, d)$ is a layered directed acyclic multi-graph (U, A) with arc labels d that encode values of the variables. 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 r and the terminal t, respectively. Each

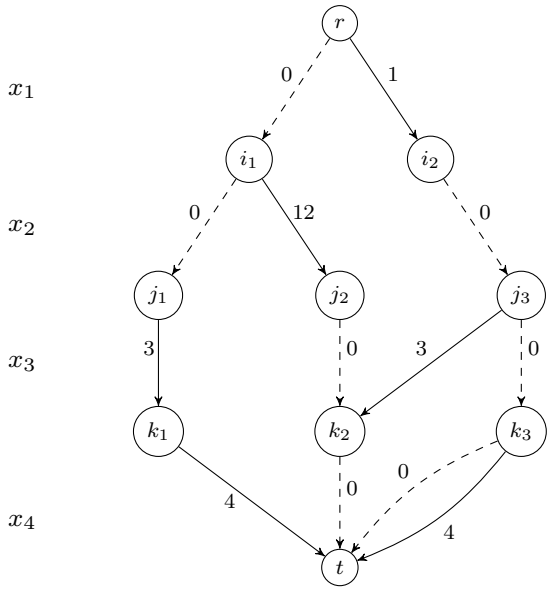


Figure 3.1: Exact BDD for the knapsack instance (3.2) in Example 1. Dashed and solid arcs represent arc labels 0 and 1, respectively. The numbers on the arcs indicate their length.

arc $a \in A$ is directed from a node in some L_j to a node in L_{j+1} and has a label $d(a) \in D_j$ that represents the value of a variable x_j. No two arcs leaving the same node have the same label, which means every node has a maximum out-degree of $|D_j|$. If all variables are binaries, then the DD is a *binary decision diagram* (BDD), which has been the subject to the majority of studies in the area due to the applications in Boolean logic [98, 89, 32]. On the other hand, a *multi-valued decision diagram* (MDD) allows out-degrees higher than 2 and therefore encodes values of general finite-domain variables. Finally, the *width* $|L_j|$ of layer L_j is the number of nodes in the layer, and the *width* of a DD B is $\max_j\{|L_j|\}$. The *size* $|B|$ of B is the number of nodes in B.

Figure 3.1 shows an exact weighted DD B for the knapsack instance (3.2) described in Example 1. It is composed of 5 layers, since the problem has 4 variables. It is a BDD because all variable domains are binaries: 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., $d((j_1, k_1)) = 1$. The width of B is 3, since L_2 and L_3 have three nodes, in particular $L_2 = \{j_1, j_2, j_3\}$. The size of the BDD B is $|B| = 10$.

Every arc-specified path $p = (a^{(1)}, \dots, a^{(n)})$ from r to t encodes an assignment to the variables $x_1, \dots, x_n$, namely $x_j = d(a^{(j)})$ for $j = 1, \dots, n$. We will denote this assignment by x^p. For example, in Figure 3.1 the arc-specified path $p = ((r, i_1), (i_1, j_1), (j_1, k_1), (k_1, t))$ encodes the assignment $x^p = (0, 0, 1, 1)$. The set of r–t paths of B represents the set of assignments $\mathrm{Sol}(B)$.

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 $p = \left(a^{(1)}, \dots, a^{(k)}\right)$ rooted at r corresponds to $v(p) = \sum_{j=1}^{k} v(a^{(j)})$. A weighted DD B represents an optimization problem $\mathcal{P}$ in a straightforward way. Namely, B is an *exact decision diagram representation* of $\mathcal{P}$ if the r–t paths in B encode

precisely the feasible solutions of $\mathcal{P}$, and the length of a path is the objective function value of the corresponding solution. More formally, we say that B is *exact* for $\mathcal{P}$ when

$$\text{Sol}(\mathcal{P}) = \text{Sol}(B) \tag{E-1}$$

$$f(x^p) = v(p), \text{ for all } r\text{–}t \text{ paths } p \text{ in } B \tag{E-2}$$

In Figure 3.1 the length $v(a)$ is represented by a number above each arc a. One can verify that the BDD B depicted in this figure satisfies both conditions (E-1) and (E-2) for the knapsack problem instance (3.2). For example, the path $p = ((r, i_1), (i_1, j_1), (j_1, k_1), (k_1, t))$ is such that $v(p) = 7$, which coincides with $f(x^p) = f((0, 0, 1, 1)) = 7$.

An exact DD reduces discrete optimization to a longest-path problem. If p is a longest path in a DD B that is exact for $\mathcal{P}$, then x^p is an optimal solution of $\mathcal{P}$, and its length $v(p)$ is the optimal value $z^*(\mathcal{P}) = f(x^p)$ of $\mathcal{P}$. For Figure 3.1, the longest path is given by the path $p^* = ((r, i_1), (i_1, j_2), (j_2, k_2), (k_2, t))$ with a length of $v(p^*) = 12 = z^*$, representing the optimal solution $x^{p^*} = (0, 1, 0, 0)$.

It is common in the DD literature to allow various types of *long arcs* that skip one or more layers [32, 104]. 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 dissertation. DDs also typically have two terminal nodes, corresponding to true and false, but for our purposes only a true node is required as the terminus for feasible paths.

3.3 Construction by Top-Down Compilation

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

3.3.1 Dynamic Programming Concepts

Dynamic programming (DP) is a *recursive* optimization method. A DP model for a discrete optimization problem $\mathcal{P}$ is formulated to be solved in stages, each representing the transition from a particular system *state* to the next until a final (or *terminal*) state is reached. These transitions are controlled by the variable assignments and incur a particular cost, representing the objective function of $\mathcal{P}$. DP methods thus enumerate states instead of variable assignments, which may be potentially fewer in number.

To demonstrate the key ingredients involved in a DP model, we reproduce here the example described in [86] and show how to reformulate the 0/1 knapsack problem as a DP model. Recall the original linear model (3.1) in Example 1, where we wish to maximize the total value of the selected items, $\sum_{j=1}^{n} v_j x_j$, subject to a weight capacity constraint $L \leq \sum_{j=1}^{n} w_j x_j \leq U$ and binary domains, $x_j \in \{0,1\}$ for all j.

Since a DP is solved in stages, the first step towards formulating a DP model for the knapsack problem is to introduce a *state space*, so that transitions from one stage to another are represented as transitions between states of the system. In particular, the first stage has a pre-defined origin state denoted by *root state* $\hat{r}$. The transitions from a state to another are governed by the variables $x_1, \ldots, x_n$, which are regarded in DP as *controls*. Given a state s^j in stage j, the assignment of the control x_j defines the next state s^{j+1} that is reached.

For the knapsack problem, 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 $s^1 = 0$. In stage i, 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 $L \leq s^{n+1} \leq U$.

Finally, the objective function is represented by costs incurred by each transition. For the knapsack, we assign a cost of 0 if $x_j = 0$, and v_j otherwise. We wish to find a set of transitions that lead us from the root state to the terminal state with maximum transition cost. 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.

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} v_j x_j \\ & s^{j+1} = s^j + w_j\, x_j, \;\; j = 1, \ldots, n \\ & x_j \in \{0,1\}, \;\; j = 1, \ldots, n \\ & s^1 = 0, \; L \leq s^{n+1} \leq U \end{aligned} \tag{3.3}$$

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.

As can be perceived from the knapsack example, the main components of a DP model are the

states, the way how the controls govern the transitions, and finally the costs of each transition. To specify this formally, a DP model for a given problem $\mathcal{P}$ with n variables having domains $D_1, \ldots, D_n$ 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* $\hat{t_1}, \hat{t_2}, \ldots, \hat{t_k}$. To facilitate notation, we also consider an *infeasible state* $\hat{0}$ that leads to infeasible solutions to $\mathcal{P}$. The state space is partitioned into sets for each of the $n+1$ stages, i.e. S is the union of the sets $S_1, \ldots, S_{n+1}$, where $S_1 = \{\hat{r}\}$, $S_{n+1} = \{\hat{t_1}, \ldots, \hat{t_k}, \hat{0}\}$, and $\hat{0} \in S_j$ for $j = 2, \ldots, n$.

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 $j = 1, \ldots, n$. Also, a transition from an infeasible state always lead 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 $j = 1, \ldots, n$.

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 $(s, x) = (s^1, \ldots, s^{n+1}, x_1, \ldots, x_n)$ and is written

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

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

$$s^{n+1} = \hat{t} \text{ and } \hat{f}(s,x) = f(x), \text{ if } x \text{ is feasible for } \mathcal{P} \tag{A1}$$

$$s^{n+1} = \hat{0}, \text{ if } x \text{ is infeasible for } \mathcal{P} \tag{A2}$$

3.3.2 Top-Down Compilation

We now show how to compile a DD based on a DP model in a top-down fashion. The construction of an exact weighted decision diagram from a DP formulation (DP) 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 construction procedure is stated as Algorithm 1, described as follows. Begin with the root node 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 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

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 t

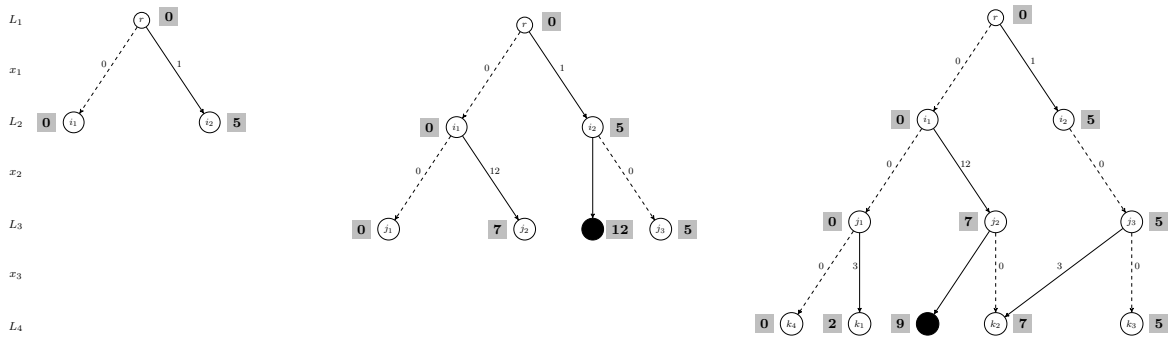
equal to the transition cost. At the last stage, identify all terminal states $\hat{t_1}, \ldots, \hat{t_k}$ to a terminal node t. Notice that, because distinct nodes always have distinct states, the algorithm identifies each node with the state associated with that node.

EXAMPLE 2 Figure 3.2 depicts three consecutive iterations of Algorithm 1 for the 0/1 knapsack problem instance (3.2). The DP states from model (3.3) are represented as grey boxes next to the nodes that identify them. In the first iteration, presented in Figure 3.2a, layer L_1 is built with the root node r having state 0, and layer L_2 is built with nodes i_1 and i_2 having states 0 and 5, respectively. In the second iteration, layer L_3 is built with four nodes as Figure 3.2b shows. However, since all costs are positive, we can already identify that the black-filled node with state 12 is infeasible (i.e., $\hat{0}$), as all of its branches will have a state larger than the maximum weight capacity of 8. This node is then removed from the BDD. Figure 3.2c presents one more iteration, where layer L_4 is built. As before, we can already remove node with state 9. Notice that a dashed arc leaving j_2 and a solid arc leaving j_3 have the same state 7, so they are directed to the same node k_2.

Finally, we note that node k_4 will be removed in the next iteration of the Algorithm, since all of its branches will be infeasible (as the resulting total weight will be less than 5), yielding the exact BDD represented in Figure 3.1. □

Algorithm 1 assumes that the controls $x_1, \ldots, x_n$ are ordered according to the DP model input. Nevertheless, as studied in [15], 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 related to the combinatorial structure of the problem that the DD represents. We present a study case on this relationship in Section 3.5. In particular, a follow-up work of our study shows that the size of a DD for set packing and set covering problems are related to the bandwidth of the constraint matrix that results from this ordering [80].

The outlined DD construction procedure can also be perceived as a particular representation of the *state-graph* of the DP formulation [87]. Suppose that (DP) is valid for problem $\mathcal{P}$, and consider the state-transition graph for (DP). Omit all occurrences of the infeasible state $\hat{0}$, and let each



(a) L_1 and L_2. (b) L_3. (c) L_4.

Figure 3.2: Three consecutive iterations of Algorithm 1 for the 0/1 knapsack problem (3.2). Grey boxes correspond to the DP states in model (3.3), and black-filled nodes indicate infeasible nodes.

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 multi-graph B_{DP} is an exact DD for $\mathcal{P}$, because paths from state r to state t in B_{DP} correspond precisely to feasible solutions of (DP), and the objective function value of the corresponding solution is the path length.

We remark in passing that the resulting DD is not necessarily *reduced* [32, 132], meaning that a layer j may contain two or more *equivalent* nodes. Two nodes are equivalent when the paths from each to t correspond to the same set of assignments to $(x_j, \ldots, x_n)$. In a reduced DD, all equivalent nodes in a layer are superimposed. 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 3.3.7.

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 3.3.9.

3.3.3 Maximum Independent Set Problem

A discrete optimization problem that will be studied throughout this dissertation is the *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 Figure 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 [53], bioinformatics [51], and social network analysis [13].

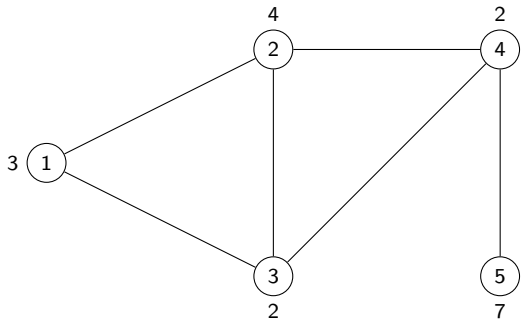


Figure 3.3: Example of a graph with vertex weights for the MISP. Vertices are assumed to be labeled arbitrarily, and the number above each circle indicates the vertex weight.

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

$$\begin{aligned} \max \ & \sum_{j=1}^{n} w_j x_j \\ & x_i + x_j \leq 1, \ \text{ for all } (i,j) \in E \\ & x_j \in \{0,1\}, \ \text{ for all } j \in V \end{aligned} \tag{3.4}$$

In the formulation above, we define a variable x_j for each vertex $j \in V$ with binary domain $D_j = \{0,1\}$, 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 edge from being selected simultaneously, i.e. $\mathcal{C} = \{x_i + x_j \leq 1 \mid (i,j) \in E\}$. For the graph in Figure 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, $\text{Sol}(\mathcal{P})$ is defined by the vectors x that represents the family of independent sets $V \cup \{\{1,4\},\{1,5\},\{2,5\},\{3,5\}\}$.

Example 3 Figure 3.4 shows an exact weighted BDD B for the MISP problem defined over the graph G in Figure 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 $\text{Sol}(\mathcal{P}) = \text{Sol}(B)$. 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 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

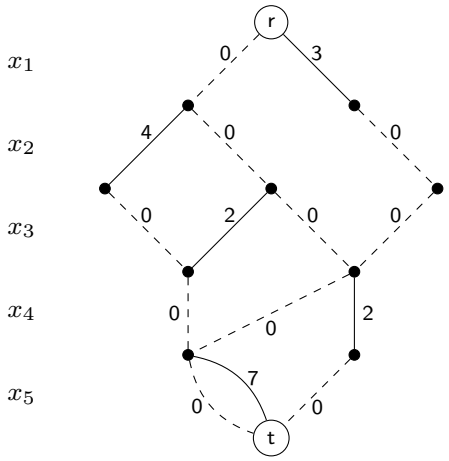


Figure 3.4: Exact BDD for the MISP on the graph in Figure 3.3. Dashed and solid arcs represent labels 0 and 1, respectively.

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 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 thus 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 Figure 3.3. The states associated with nodes of B_{DP} are shown in Figure 3.5. For example, node u_1 has state $\{2, 3, 4, 5\}$, representing the vertex set $V \setminus \{1\}$. We will show in Section 3.5 that the state space described above yields a *reduced DD*, thus it is the smallest possible DD for the control ordering given as input.

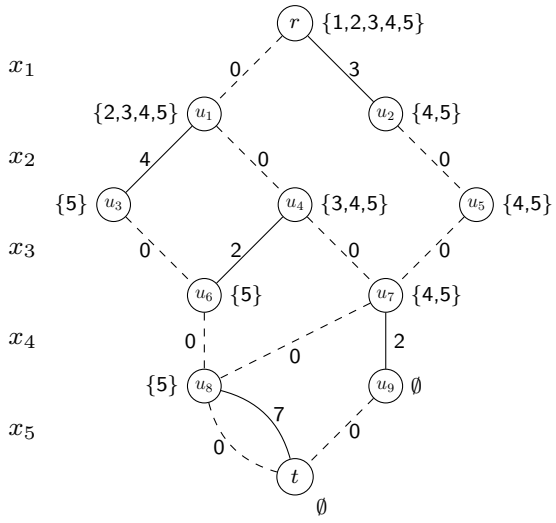


Figure 3.5: Exact BDD with states for the MISP on the graph in Figure 3.3.

3.3.4 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-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 [62].

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, $C_i = \{j \,|\, a_{i,j} = 1\}$, and let $\texttt{last}(C_i) = \max\{j \,|\, j \in C_i\}$ 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.

EXAMPLE 4 Consider the SCP instance with

$$c = (2, 1, 4, 3, 4, 3)$$

and

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

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 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. We show this in Example 5.

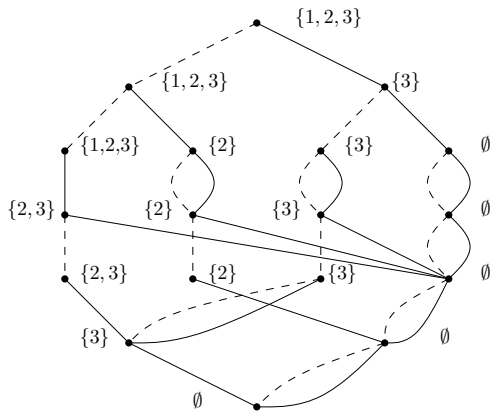


Figure 3.6: Exact BDD for the SCP instance in Example 4.

EXAMPLE 5 Consider the set covering problem

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

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 $\tilde{x} = (1)$. $\square$

There are several ways to modify the state function so that the resulting DD is reduced, as presented in [26]. This requires only polynomial time to compute per partial solution, but nonetheless at an additional computational cost.

3.3.5 Set Packing Problem

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

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

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 one 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 one: $S_j = 2^{\{1,\ldots,m\}} \cup \{\hat{0}\}$ for $j = 2, \ldots, n$. Initially, $\hat{r} = \{1, \ldots, m\}$. 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

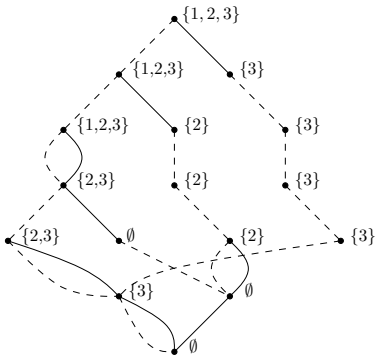


Figure 3.7: Exact reduced BDD for the SPP instance in Example 6.

$(i \in A_j)$ 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.

Example 6 Consider the SPP instance with the same constraint matrix A as in Example 4, but with weight vector

$$c = (1, 1, 1, 1, 1, 1).$$

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 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. This is shown in the following example.

EXAMPLE 7 Consider the problem

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

and the two partial solutions $x^1 = (1, 0)$, $x^2 = (0, 1)$. We have 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 3.3.3 , which we demonstrate to have this property in Section 3.5.

3.3.6 Single Machine Makespan Minimization

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

For example, consider the following variant of a *single machine makespan minimization problem* (MMP) [111]. Let $\mathcal{J}$ represent a set of n jobs that must be scheduled in 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 3.1 depicts an instance of the MMP problem with 3 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* [108]. More complex position position-dependent problems usually represent machine deterioration, and literature on this topic is relatively new [2].

To formulate the MMP as an optimization problem, we let x_i represent the i-th job to be

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

Table 3.1: Processing times of a single machine makespan minimization problem. Rows and columns represent the job index and the position in the schedule, respectively.

processed in the machine. The MMP can be written as

$$\min \sum_{i=1}^{n} p_{i,x_i}$$

$$\textsc{Alldifferent}(x_1, \ldots, x_n) \tag{3.5}$$

$$x_i \in \{1, \ldots, n\}, \quad i = 1, \ldots, n \tag{3.6}$$

Constraint (3.5) is typical in constraint programming models [119] and indicates that variables $x_1, \ldots, x_n$ must assume pairwise distinct values, i.e. they define a permutation of $\mathcal{J}$. Hence, the set of feasible solutions to the MMP is the set of permutation vectors of $\mathcal{J}$.

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 in 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 in 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.

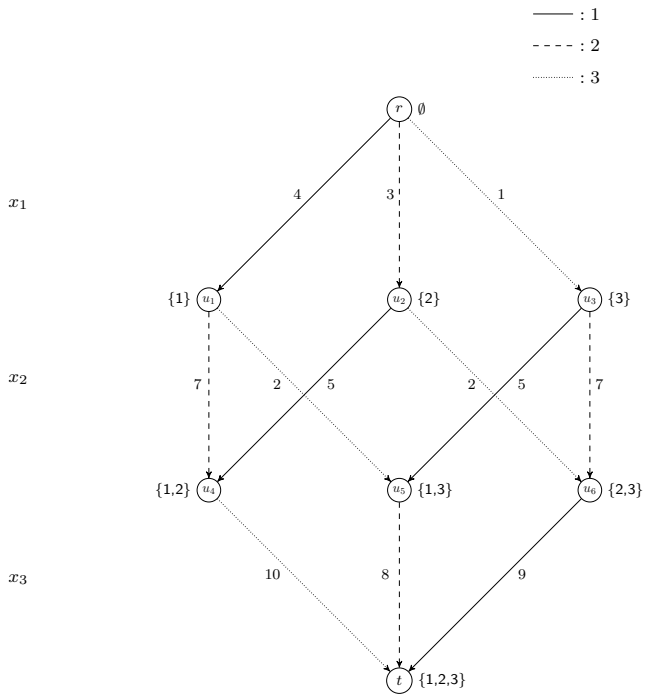


Figure 3.8: Example of an MDD for the minimum makespan problem in Table 3.1. Solid, dashed, and dotted arcs represent labels 1, 2, and 3, respectively.

EXAMPLE 8 Figure 3.8 depicts the MDD with node states for the MMP instance defined in Table 3.1. 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. □

Existing works focus on representation issues of the set of permutation vectors (e.g., [122, 131, 11]). DD representations of permutations have also been suggested in the literature [105]. The DP model presented here yields the same DD as in [78]. It will be again the subject of our discussion in Section 4.3.1. Also, in Chapter 7 we show how to minimize different scheduling objective functions over the same decision diagram representation.

3.3.7 Maximum Cut Problem

Given a graph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$, 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 VLSI design, statistical physics, and other problems [79, 55].

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

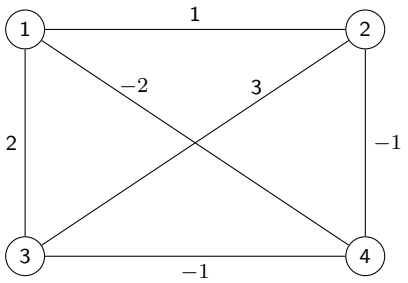


Figure 3.9: Graph with edge weights for the MCP.

$T(x) = \{j \mid x_j = \mathrm{T}\}$, the objective function is $f(x) = v(S(x), T(x))$. Since any partition is feasible, $\mathcal{C} = \emptyset$. Thus the MCP can be written as

$$\begin{aligned} \max \;\; & v(S(x), T(x)) \\ & x_j \in \{S, T\}, \;\; j = 1, \ldots, n \end{aligned} \tag{3.7}$$

Example 9 Consider the graph G depicted in Figure 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 $j, \ldots, n$, 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.

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

$$s^{k+1}_\ell = \left\{ \begin{array}{ll} s^k_\ell + w_{k\ell}, & \text{if } x_k = \mathrm{S} \\ s^k_\ell - w_{k\ell}, & \text{if } x_k = \mathrm{T} \end{array} \right\}, \;\; \ell = k+1, \ldots, n$$

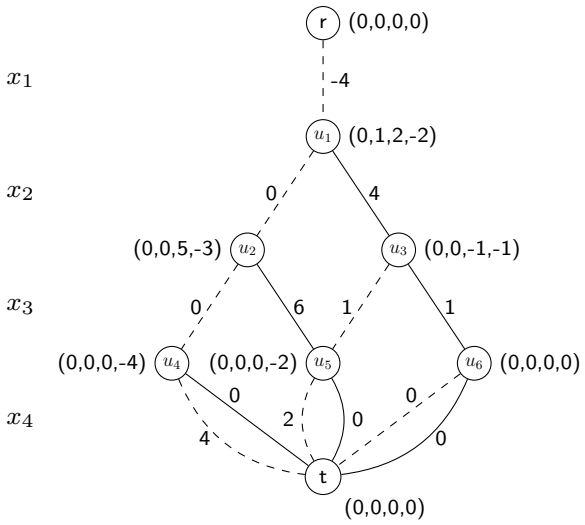


Figure 3.10: Exact BDD with states for the MCP on the graph in Figure 3.9

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

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

- 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(a) shows the resulting weighted BDD for the example discussed earlier.

Example 10 Consider the graph G in Figure 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 $(S, T) = (\{1, 2, 4\}, \{3\})$. $\square$

3.3.8 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 2 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 [90, 95, 45].

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 $\mathcal{C}$ is empty, and 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

$$\begin{aligned} \max \ & \sum_{i=1}^{m} w_i c_i(x) \\ & x_j \in \{\mathrm{F}, \mathrm{T}\}, \quad j = 1, \ldots, n \end{aligned} \tag{3.8}$$

EXAMPLE 11 Consider the following instance of MAX-2SAT with three boolean variables x_1, x_2, and x_3:

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 optimal solution to the instance above 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 $\mathcal{C}$ 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 $j, k \in \{1, \ldots, n\}$ 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 $(s_1^k, \ldots, s_n^k)$ 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: $S_k = \left\{ s^k \in \mathbb{R}^n \mid s_j^k = 0,\ j = 1, \ldots, k-1 \right\}$, with root state and terminal state equal to $(0, \ldots, 0)$
- transition functions: $t_k(s^k, x_k) = (0, \ldots, 0, s_{k+1}^{k+1}, \ldots, s_n^{k+1})$, where

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

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

$$h_k(s^k, x_k) = \left\{ \begin{array}{ll} (-s_k^k)^+ + \displaystyle\sum_{\ell > k} \left(w_{k\ell}^{\mathrm{FF}} + w_{k\ell}^{\mathrm{FT}} + \min\left\{ (s_\ell^k)^+ + w_{k\ell}^{\mathrm{TT}}, (-s_\ell^k)^+ + w_{k\ell}^{\mathrm{TF}} \right\} \right), & \text{if } x_k = \mathrm{F} \\ (s_k^k)^+ + \displaystyle\sum_{\ell > k} \left(w_{k\ell}^{\mathrm{TF}} + w_{k\ell}^{\mathrm{TT}} + \min\left\{ (s_\ell^k)^+ + w_{k\ell}^{\mathrm{FT}}, (-s_\ell^k)^+ + w_{k\ell}^{\mathrm{FF}} \right\} \right), & \text{if } x_k = \mathrm{T} \end{array} \right\},$$
$$k = 2, \ldots, n$$

- root value: $v_r = 0$

Example 12 Figure 3.11(a) shows the resulting states and transition costs for the MAX-2SAT instance in Example 11. Notice that the longest path p yields the solution $x^p = (\mathrm{F}, \mathrm{T}, \mathrm{T})$ with length 14. □

3.3.9 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 [21], the proof of correctness of the SCP and the SPP formulations in [25], and finally the proof of correctness of the MMP formulation in [41].

Theorem 1 *The specifications in Section 3.3.7 yield a valid DP formulation of the MCP.*

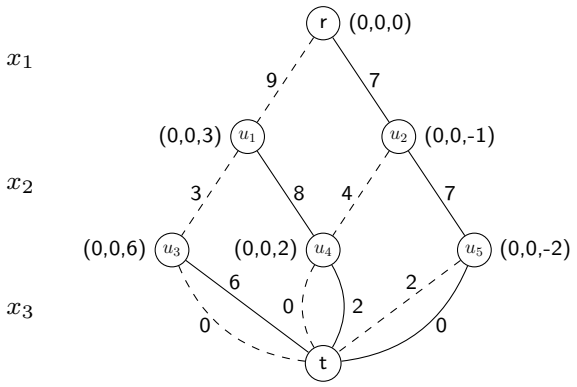


Figure 3.11: Exact BDD with states for the MAX-2SAT problem in Example 11.

Proof. Note that any solution $x \in \{S,T\}^n$ is feasible so that we need only show that condition (A1) 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 $(\bar{s},\bar{x})$ be an arbitrary solution of (DP), it remains to show that $\hat{f}(\bar{s},\bar{x}) = f(\bar{x})$. 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

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

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

$$s^k_\ell = L^\ell_{k-1} - R^\ell_{k-1}, \ \text{ for } \ell \ge k \tag{Sk}$$

where

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

We will show the following inductively:

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

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 (Hk) implies (Hn) when $k = n$.

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

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

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

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

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

$$\begin{aligned} &\sum_{\ell \geq k} \min\left\{L_{k-1}^\ell, R_{k-1}^\ell\right\} - N_{k-1} + (\sigma_k(L_{k-1}^k - R_{k-1}^k))^+ + \sum_{\substack{\ell>k \\ \sigma_k(L_{k-1}^\ell - R_{k-1}^\ell) w_{k\ell} \geq 0}} \min\left\{|L_{k-1}^\ell - R_{k-1}^\ell|, |w_{k\ell}|\right\} \\ &\quad = \sum_{\substack{j<k \\ \bar{x}_j \neq \bar{x}_k}} w_{jk} + \sum_{\ell>k} \min\left\{L_k^\ell, R_k^\ell\right\} - N_k \end{aligned} \tag{Eq1}$$

We will show that (Eq1) holds when $\bar{x}_k = \mathrm{T}$. The proof for $\bar{x}_k = \mathrm{S}$ is analogous. Using the fact that $R_k^\ell = R_{k-1}^\ell + w_{k\ell}$, (Eq1) can be written

$$\begin{aligned} &\min\left\{L_{k-1}^k, R_{k-1}^k\right\} + \sum_{\ell>k} \min\left\{L_{k-1}^\ell, R_{k-1}^\ell\right\} + (L_{k-1}^k - R_{k-1}^k)^+ \\ &\qquad\qquad + \sum_{\substack{\ell>k \\ (L_{k-1}^\ell - R_{k-1}^\ell) w_{k\ell} \geq 0}} \min\left\{|L_{k-1}^\ell - R_{k-1}^\ell|, |w_{k\ell}|\right\} \\ &\quad = L_{k-1}^k + \sum_{\ell>k} \min\left\{L_{k-1}^\ell, R_{k-1}^\ell + w_{k\ell}\right\} - (N_k - N_{k-1}) \end{aligned} \tag{Eq2}$$

The first and third terms of the left-hand side of (Eq2) sum to L_{k-1}^k. We can therefore establish (Eq2) by showing that for each $\ell \in \{k+1, \ldots, n\}$, we have

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

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 2 *The specifications in Section 3.3.8 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 (DP), 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

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

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

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

We will show the following inductively:

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

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

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

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

where σ_k is 1 if $\bar{x}_k = \mathrm{T}$ 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

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

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

to show

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

The second and third terms of the left-hand side of (Eq-SAT) sum to L_{k-1}^k. Also

$$\mathrm{SAT}_k(\bar{x}) - \mathrm{SAT}_{k-1}(\bar{x}) = L_{k-1}^k + \sum_{\ell>k}\left(w_{k\ell}^{\mathrm{TF}} + w_{k\ell}^{\mathrm{TT}}\right)$$

We can therefore establish (Eq-SAT) by showing that

$$\begin{aligned}
&\min\left\{L_{k-1}^\ell, R_{k-1}^\ell\right\} + \min\left\{(L_{k-1}^\ell - R_{k-1}^\ell)^+ + w_{k\ell}^{\mathrm{FT}}, (R_{k-1}^\ell - L_{k-1}^\ell)^+ + w_{k\ell}^{\mathrm{FF}}\right\} \\
&\qquad = \min\left\{L_{k-1}^\ell + w_{k\ell}^{\mathrm{FT}}, R_{k-1}^\ell + w_{k\ell}^{\mathrm{FF}}\right\}
\end{aligned}$$

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

3.4 Construction by Separation

Construction by 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, 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 consider 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 can be either general (e.g., separate arbitrary variable assignments) or may exploit problem structure.

Given a discrete optimization problem $\mathcal{P}$, the method starts with a DD B' that is a *relaxation* of the exact DD B: $\mathrm{Sol}(B') \supseteq \mathrm{Sol}(B) = \mathrm{Sol}(\mathcal{P})$. That is, B' encodes all the feasible solutions to $\mathcal{P}$ 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 $\mathcal{P}$ (if one is only interested in the optimality) This separation method was first introduced by [78] for building approximate DDs in constraint programming models. The procedure in the original work, denoted by *incremental*

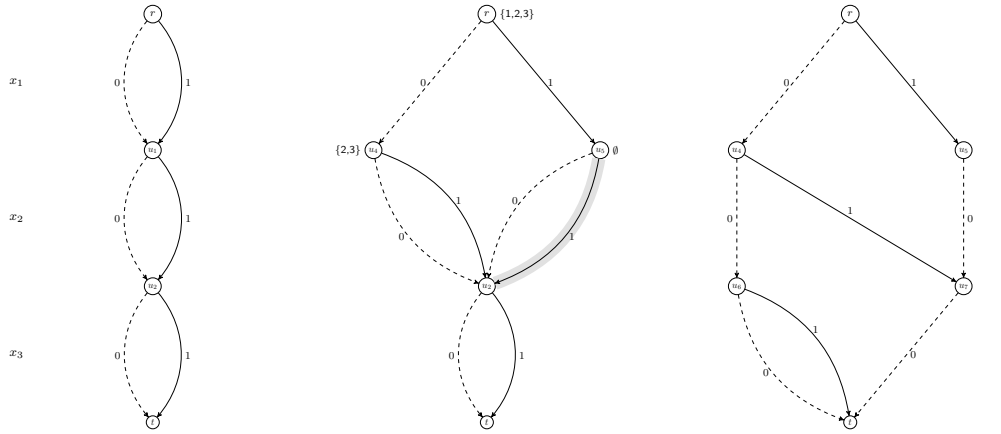
Algorithm 2 Exact DD Compilation by Separation

Let $B' = (U', A')$ be a DD such that $\text{Sol}(B') \supseteq \text{Sol}(\mathcal{P})$.
while $\exists$ constraint C violated by B' **do**
  Let $s(u) = \chi$ for all nodes $u \in B'$
  $s(u) := \hat{r}$
  for $j = 1$ to n **do**
    for $u \in L_j$ **do**
      for each arc $a = (u, v)$ leaving node u **do**
        if $t_j^C(s(u), d(a)) \neq \hat{0}$ **then**
          Remove arc a from B
        else if $s(v) = \chi$ **then**
          $s(v) = t_j^C(s(u), d(a))$
        else if $s(v) \neq t_j^C(s(u), d(a))$ **then**
          Remove arc (u, v)
          Create new node v' with $s(v') = t_j^C(u, d(a))$
          Add arc (u, v')
          Copy outgoing arcs from v as outgoing arcs from v'
          $L_j := L_j \cup \{v'\}$

refinement, is slightly different than the presented here and will be described in Section 4.3.

The outline of the method is depicted in Algorithm 2. The algorithm starts with a DD B' that is a relaxation of $\mathcal{P}$ and finds a constraint that is potentially violated by paths in B'. Such constraint could be obtained by iterating on the original set of constraints of $\mathcal{P}$, 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 re-set 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 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 to 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 $\mathcal{P}$.



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

Figure 3.12: First three iterations of the separation method for the problem in Example 13.

EXAMPLE 13 Consider the following optimization problem $\mathcal{P}$:

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

We partition the constraints into two sets:

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

We will compile the exact BDD for $\mathcal{P}$ 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 $\mathcal{P}$ as input. This can be trivially obtained by creating an 1-width BDD that contains the Cartesian product of the variable domains, as depicted in Figure 3.12a. The arc lengths were already 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 3.3.3 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 leads

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 Figure 3.12b, 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 Figure 3.12b), therefore it will be removed when processing layer L_2.

The resulting DD after separating constraint C_1 is presented in Figure 3.12c. 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. $\square$

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 [132]. However, we 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 some structured cases, as we will describe in Section 3.4.1.

There are some potential benefits of this procedure over the top-down approach from Section 3.3. 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 [86]). Finally, the DD B' that contains a feasible longest path could be potentially much smaller than the exact DD B for $\mathcal{P}$.

As in the top-down approach of Section 3.3, 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 [132].

3.4.1 Separating Partial Assignments: Growth of DDs

A key issue for separating DDs is how fast they grow as solutions are separated. In IP, a solution can be separated with a single inequality, so that the continuous relaxation grows only linearly with the number of solutions separated. We will show, however, that a separating DD can grow exponentially with the number of solutions separated, even if the DD is reduced.

To this end, we consider the problem of separating *partial assignments* on binary decision diagrams. Namely, assume all decision variables x are binaries, and let $I \subseteq \{1, \dots, \}$ be a subset of variable indices. Also, let $x_i = \bar{x}_i$ for all $i \in I$ be a partial assignment of the decision variables x. The separating problem for this partial assignment and a given BDD B' is to find a BDD B''

that excludes this partial assignment from B''. That is, the r-t paths in B'' consist of those in B' except those corresponding to assignments x with $x_i = \bar{x}_i$ for $i \in I$.

As specified in Section 3.4, we need to define a suitable DP model to enforce the constraint $x_i \neq \bar{x}_i$ for $i \in I$ over all tuples x encoded by B'. This is accomplished as follows. Let a state s^j at stage j represents whether or not all previous partial assignments x' up to that stage satisfy $x'_i = \bar{x}_i$, for $i \in I \cap \{1, \ldots, j\}$, thus $s^j \in \{0, 1\}$ for all states j. We let the root state $\hat{r}$ be such that $\hat{r} = 1$. Given a domain value $d \in \{0, 1\}$, the transition function t_j is given by:

$$t_j(s^j, d) = \begin{cases} 0, & \text{if } s^j = 0 \text{ or } (j \in I \text{ and } d \neq \bar{x}_j) \\ 1, & \text{otherwise.} \end{cases} \tag{3.9}$$

Transition costs and root values are not necessary in this context.

THEOREM 3 *The state definition and transition function (3.9) are sufficient to separate constraint $x_i = \bar{x}_i$ for $i \in I$ from a BDD B' using Algorithm 2.*

EXAMPLE 14 Suppose the separation algorithm is applied to the BDD of Fig. 3.13(a) to cut off the partial assignment $(x_2, x_4) = (1, 1)$. The resulting BDD is shown in Fig. 3.13(b). Because nodes u_8^1 and u_9^1 do not lie on an r-t path, they and the two arcs adjacent to them may be dropped from the diagram. □

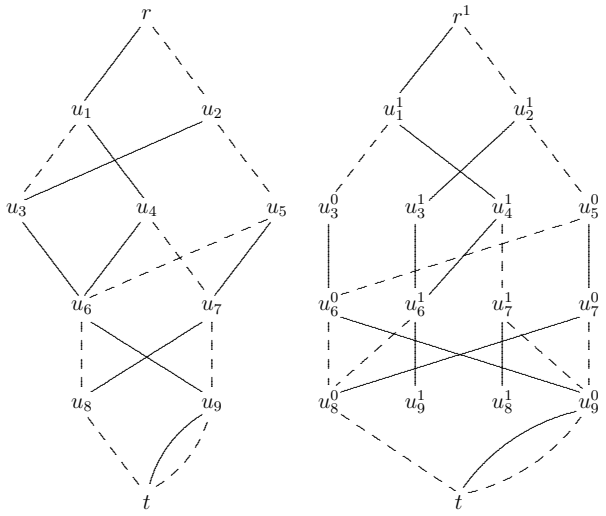
We first observe that it may be possible to reduce a separating BDD even when the original BDD is irreducible. Consider, for example, the BDD with two binary variables x_1, x_2 shown in Fig. 3.14(a). Using the algorithm to separate the partial assignment $x_2 = 1$ results in the BDD of Fig. 3.14(b), which can be reduced to the BDD of Fig. 3.14(c).

Suppose the separation algorithm is applied to BDD B to obtain BDD B'. A simple bound on the size of B' results from observing that each layer of B' contains at most two copies of the nodes in the corresponding layer of B. When a single solution is separated from B, at most node per layer can have state 1. We therefore have the following.

THEOREM 4 *The separation algorithm at most doubles number of nodes in the given BDD. When separating a single solution, it adds at most 1 node to each layer.*

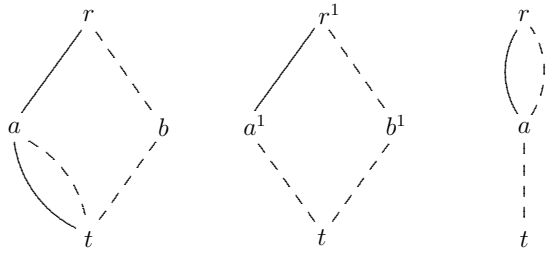
We can also state a bound on the complexity of the separation algorithm. Creating layer $i+1$ of B' requires examining at most $2|L_i|$ nodes in layer i of B', where L_i is the set of nodes in layer i of B. For each node, the two possible values of x_i are examined in the worst case. So the complexity is $\mathcal{O}(\sum_i |L_i|) = \mathcal{O}(m)$, where m is the number of nodes in B.

We can give a sharper bound on the size of B' before reduction as follows, which can be shown by induction.



(a) (b)

Figure 3.13: (a) A BDD with layers corresponding to 0-1 variables $x_1, \ldots, x_5$. (b) BDD after separation of partial assignment $(x_2, x_4) = (1, 1)$ from (a).



(a) (b) (c)

Figure 3.14: (a) Initial BDD, (b) BDD after separation of partial assignment $x_2 = 1$, and (c) reduced version of (b).

THEOREM 5 *Suppose the separation algorithm is applied to a BDD B to obtain B'. Let k be the smallest index and ℓ the largest index in I. Then the number of nodes in layer i of B' is bounded above by U_i, where*

$$U_i = \begin{cases} |L_i| & \text{for } i = 1, \ldots, k \\ |L_i| + \psi_i & \text{for } i = k+1, \ldots, \ell \\ |L_i| & \text{for } i = \ell+1, \ldots, n \end{cases}$$

The quantities ψ_i are given by $\psi_k = |L_k|$, and

$$\psi_i = \begin{cases} \min\{|L_i|, \psi_{i-1}\} & \text{if } i-1 \in I \\ \min\{|L_i|, 2\psi_{i-1}\} & \text{otherwise} \end{cases} \tag{3.10}$$

for $i = k+1, \ldots \ell$.

An interesting special case is $I \subset \{1, \ldots, \ell\}$, so that the partial assignment to be separated involves only a subset of the first ℓ variables. In this case, nodes are added only to the first ℓ layers of B to obtain B'. No nodes are added to the remainder of the BDD.

COROLLARY 6 *If* $I \subset \{1, \ldots, \ell\}$, *then layer* i *of* B' *contains at most* $|L_i|$ *nodes for* $i = \ell+1, \ldots, n$. *In particular, if* $I = \{1, \ldots, \ell\}$, *then layer* i *of* B' *contains at most* $|L_i|+1$ *nodes for* $i = 1, \ldots, \ell$ *and at most* $|L_i|$ *nodes for* $i = \ell+1, \ldots, n$.

The bound in Theorem 5 grows exponentially with the number of partial assignments separated. We might ask whether the separating BDD can itself grow exponentially. One might expect that it can, because otherwise one can solve the set covering problem in polynomial time, which implies $P = NP$. This is because a set covering constraint can be written $\sum_{i \in I} x_i \geq 1$, which is equivalent to excluding the solutions in $\bar{x}(I)$ where $\bar{x}_i = 0$ for $i \in I$. Thus we can let B be the diagram representing all 0-1 tuples x, and B' the diagram that separates all the set covering constraints. The set covering problem, which minimizes $\sum_i x_i$, can now be solved by placing a cost of 1 on all 1-arcs and 0 on all 0-arcs, and finding a shortest path in B'.

THEOREM 7 *Given a BDD* B, *the number of nodes in the reduced BDD that separates the solutions in* $\bar{x}^1(I_1), \ldots, \bar{x}^m(I_m)$ *from* B *can grow exponentially with respect to* m *and the size of* B.

The theorem is proved by letting B be a BDD of width 1 that represents all possible 0-1 solutions, and then separating the partial assignments $(x_1, x_n) = (1,1)$, $(x_2, x_{n-1}) = (1,1)$, $(x_3, x_{n-2}) = (1,1)$, etc. The separating BDD for $n = 6$ appears in Fig. 3.15. The BDD grows exponentially with n and is reduced as well. As it happens, the above separation algorithm yields this reduced BDD.

3.5 Variable Ordering: MISP Study Case

The construction algorithms in Sections 3.3 and 3.4 assume that the ordering of the decision variables in the DD is defined according to the discrete model input. However, it is well-known from the application of BDDs for Boolean functions [132] and knapsack problems [16] that different variable orderings for the same problem instance may differ drastically in their sizes.

In this section we provide the first analysis on 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. Such analysis is a critical question in the area of decision diagrams for optimization. For example, the variable ordering deeply influences the optimization bounds obtained from approximate DDs, as shown in Section 4.2.4. The overall analysis presented here could possibly be extended to other problem classes as well, as presented in a follow-up work to our study which further analyses the orderings for set covering and set packing problems [80].

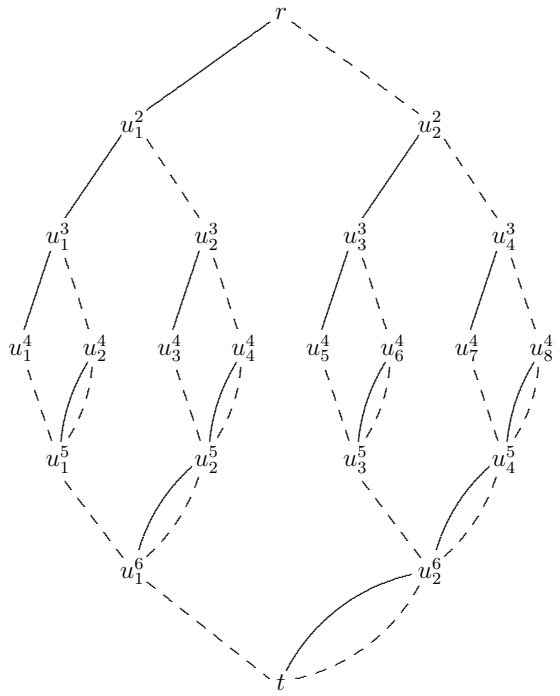
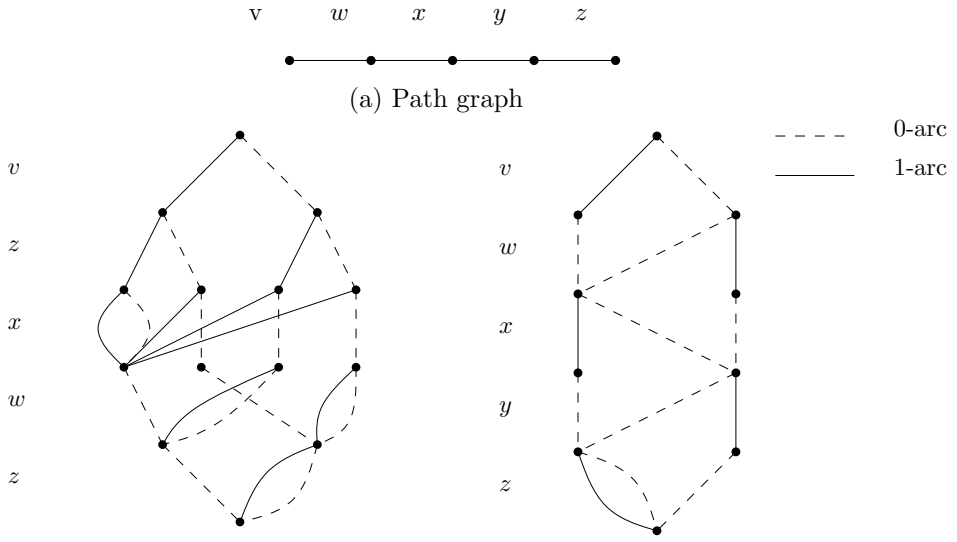


Figure 3.15: BDD that excludes partial assignments $x = (1, \cdot, \cdot, \cdot, \cdot, 1)$, $(\cdot, 1, \cdot, \cdot, 1, \cdot)$, and $(\cdot, \cdot, 1, 1, \cdot, \cdot)$.

We will particularly focus on the maximum independent set problem (MISP) for our analysis, first described here in Section 3.3.3. 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 associate weights with each vertex $j \in V$, the MISP asks for a maximum-weight independent set of G.

For the MISP, changing the variable ordering corresponds to relabeling the indices of vertices in G (or, equivalently, building the decision diagram on an isomorphic graph of G). The order of variables plays a key role in the size of the exact BDDs for the MISP. The impact of different orderings can be substantial, as shown in Figure 3.16. The example demonstrates two orderings for the graph presented in Figure 3.16a. The first ordering is constructed by alternating between the endpoints of the path, yielding a BDD of width 4 as depicted in Figure 3.16b. If vertices are taken according to the path order, the exact BDD has half the width, as presented in Figure 3.16c.

We will now analyze variable orderings for the BDD representing the family of independent sets of a problem. We first examine particular classes of graphs, namely cliques, paths, and trees. We establish polynomial bounds on the widths (and hence size) of the exact BDDs with respect to the graph size. This is achieved by providing an ordering of the vertices that forces the width to be within a certain bound. Finally, we discuss the width for general graphs.



(a) Path graph

(b) Ordering with width 4

(c) Ordering with width 2

Figure 3.16: Graph and exact BDD for two different orderings.

3.5.1 Notation

For a graph $G = (V, E)$, let $\mathcal{I}(G)$ be the family of independent sets in G. Also, let $G[W]$ be the graph induced by a subset $W \subseteq V$ and $\overline{W} := V \backslash W$. Two disjoint subsets $I, J \subset V$ are *independent* if $(w, v) \notin E$ for any $w \in I$, $v \in J$. The *neighborhood* $N(v)$ of $v \in V$ is defined as $N(v) = \{w : (w, v) \in E\}$. Let $\mathcal{I}(G)$ be the family of independent sets in G. A *partial solution with respect to* $W \subseteq V$ corresponds to any subset $I \subseteq W$, which is *feasible* if $I \in \mathcal{I}(G[W])$. Given a partial feasible solution I with respect to W, the *set of feasible completions of* I *with respect to* W is given by $C(I \mid \overline{W}) = \{J \mid J \subseteq \overline{W}, I \cup J \in \mathcal{I}(G)\}$.

A corresponding BDD $B = (U, A)$ for the MISP above defines a bijection between the vertices $v \in V$ and the layers $L_1, \ldots, L_n$; let v_j be the associated layer of vertex v, with $V_j = \{v_1, \ldots, v_j\}$. As in Section 3.3.3, with every arc-specified path $p = (a_1, \ldots, a_n)$ from the root r to the terminal t we associate a subset $I^p \subseteq V$ defined by $I^p := \{v_j : d(a_j) = 1\}$.

For a given BDD node $u \in U$, we let $B^+|_u$ be the subgraph of B induced by the subset of nodes composed of u, the root $r \in U$, and all nodes $v \in U$ lying on some directed path from r to u. In addition, we preserve the arc labels as in B; therefore, $B^+|_u$ is also a BDD. Analogously, let $B^-|_u$ be the subgraph of B induced by the subset of nodes composed by u, the terminal $t \in U$, and all nodes $v \in U$ such that there is a directed path from u to t. Also, let $B^+|_{L_j}$ be the digraph induced by $L_1, \ldots, L_j$ and similarly $B^+|_{L_j}$ be the digraph induced by $L_j, \ldots, L_{n+1}$, with $\text{Sol}(B^+|_{L_j}) = \cup_{u \in L_j} \text{Sol}(B^+|_u)$ and $\text{Sol}(B^-|_{L_j}) = \cup_{u \in L_j} \text{Sol}(B^-|_u)$.

Notice that, for a node u, any path $p = (a_1, \ldots, a_{j-1})$ in $B^+|_u$ also corresponds to a vertex subset in $G[V_{j-1}]$ and any path $p = (a_j, \ldots, a_n)$ in $B^-|_u$ corresponds to a vertex subset in $G[\overline{V}_{j-1}]$.

Moreover, each solution corresponds to at most one path in any BDD because no node has two arcs with the same label directed out of it.

3.5.2 Reduced BDDs for the MISP

A *reduced* BDD is one for which $\text{Sol}(B^-|_u) \neq \text{Sol}(B^-|_{u'})$ for any two nodes u and u' on the same layer. It can be shown that for a particular ordering of the variables, that is, how layers are mapped into variables, there is one unique reduced BDD for any set of solutions [32].

Recall from Section 3.3.3 that the state applied in the DP model for the MISP was defined by the *eligibility set*, i.e. the set of vertices that could still be added to a partial independent set that was constructed up to that stage. We now show that a BDD built using this state definition is reduced. We first establish a condition for identifying when two independent sets $I_1, I_2 \in \mathcal{I}(G[V_{j-1}])$ have the same set of feasible completions.

THEOREM 8 *Given a graph $G = (V, E)$, a subset $\{v_1, \ldots, v_{j-1}\} = V_{j-1} \subseteq V$ of the vertices of G, and two independent sets $I_1, I_2 \subseteq \mathcal{I}(G[V_{j-1}])$,*

$$C(I_1 \mid \overline{V}_{j-1}) = C(I_2 \mid \overline{V}_{j-1}) \iff \overline{V}_{j-1} \setminus \cup_{v \in I_1} N(v) = \overline{V}_{j-1} \setminus \cup_{v \in I_2} N(v).$$

Proof. For $I \in \mathcal{I}(G[V_{j-1}])$, we must have $C(I \mid \overline{V}_{j-1}) = \mathcal{I}(G[\overline{V}_{j-1} \setminus \cup_{v \in I_1} N(v)])$, since $\overline{V}_{j-1} \setminus \cup_{v \in I_1} N(v)$ is exactly the set of remaining vertices in G that are independent of I. Conversely, suppose $\overline{V}_{j-1} \setminus \cup_{v \in I_1} N(v) \neq \overline{V}_{j-1} \setminus \cup_{v \in I_2} N(v)$. Without loss of generality, suppose there exists some $w \in \overline{V}_{j-1} \setminus \cup_{v \in I_1} N(v)$ that is not in $\overline{V}_{j-1} \setminus \cup_{v \in I_2} N(v)$. Then, $w \in C(I_1 \mid \overline{V}_{j-1})$ but $w \notin C(I_1 \mid \overline{V}_{j-1})$, hence $\{v\} \cup I_1$ is an independent set while $\{v\} \cup I_2$ is not, concluding the proof. ■

COROLLARY 9 *The state space for the MISP described in Section 3.3.3 leads to an exact reduced BDD.*

Proof. It is a direct application of Theorem 8. ■

3.5.3 Variable Orderings and Bounds on BDD Sizes

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. We now show the following Theorems.

THEOREM 10 *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 11 *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 either be $\{v_j, \ldots, v_n\}$ (when $v_{j-1} \notin I$) and $\{v_{j+1}, \ldots, v_n\}$ (when $v_{j-1} \in I$). Therefore there can be at most 2 states in any given layer.

■

THEOREM 12 *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 $C_1, \ldots, C_k$ created upon deleted v from T have sizes $|C_i| \leq \frac{n}{2}$ [93]. 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 $v_1^i, \ldots, v_{|C_i|}^i$ be an ordering achieving this width.

Let the final ordering of the vertices in T be $v_1^1, \ldots, v_{|C_1|}^1, v_1^2, \ldots, v_{|C_k|}^k, v$ 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 $s \cup C_{i+1} \cup \cdots \cup C_k$ and $s \cup C_{i+1} \cup \cdots \cup C_k \cup \{v\}$, 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 $I \subseteq \{v_1^1, \ldots, v_{|C_1|}^1, v_1^2, \ldots, v_{j-1}^i\}$. All vertices in I are independent of the vertices in $C_{i+1}, \ldots, C_k$, and so $E(I \mid \{v_j^i, \ldots, v_{|C_i|}^i\} \cup C_{i+1} \cup \cdots \cup C_k) \supseteq C_{i+1} \cup \cdots \cup C_k$. 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 $\{v_j^i, \ldots, v_{|C_i|}^i\}$ that are independent of all vertices in I_i. As $v_1^i, \ldots, v_{j-1}^i$ 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, $E(I \mid \{v_j^i, \ldots, v_{|C_i|}^i\} \cup C_{i+1} \cup \cdots \cup C_k) \supseteq s \cup C_{i+1} \cup \cdots \cup C_k$. 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 $\omega_\ell \leq \frac{n}{2} \cdot 2 = n$, as desired. The layer remaining to bound is L_n, which contains $\{v\}$ and $\emptyset$. ■

THEOREM 13 *Let $G = (V, E)$ be any graph. There exists an ordering of the vertices for which $\omega_j \leq F_{j+1}$, where F_k is the* k^{th} *Fibonnaci number.*

Theorem 13 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 Fibonnaci numbers and the family of independent sets of a graph, as investigated throughout the Graph Theory literature (see for example [33, 59, 49, 134]). 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 of Theorem 13 Let $P = P^1, \ldots, P^k$, $P^i = \{v_1^1, \ldots, v_{i_k}^1\}$, 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_1^i and $v_{i_k}^i$ 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 $v_1^1, \ldots, v_{i_1}^1, v_1^2, \ldots, v_{i_k}^k$, *i.e.*, ordered by the paths and by the order that 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 exact 3 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 it's 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 thing 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, and finishing the proof.

■

Chapter 4

Relaxed Decision Diagrams

4.1 Introduction

Bounds on the optimal value are often indispensable for the practical solution of discrete optimization problems, as for example in branch-and-bound procedures. Such bounds are frequently obtained by solving a continuous relaxation of the problem, perhaps a linear programming (LP) relaxation of an integer programming model. In this section, we explore an alternative strategy of obtaining bounds from a *discrete* relaxation represented by a *relaxed* decision diagram.

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

$$\text{Sol}(\mathcal{P}) \;\subseteq\; \text{Sol}(B) \tag{Rel-1}$$

$$f(x^p) \;\leq\; v(p), \text{ for all } r\text{–}t \text{ paths } p \text{ in } B \text{ for which } x^p \in \text{Sol}(\mathcal{P}) \tag{Rel-2}$$

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

Example 15 Consider the graph and vertex weights depicted in Figure 4.1. Figure 3.4(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. 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

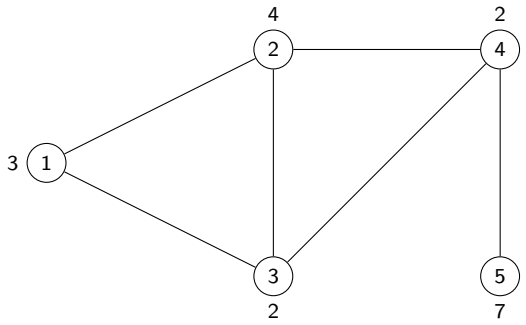


Figure 4.1: Graph with vertex weights for the MISP.

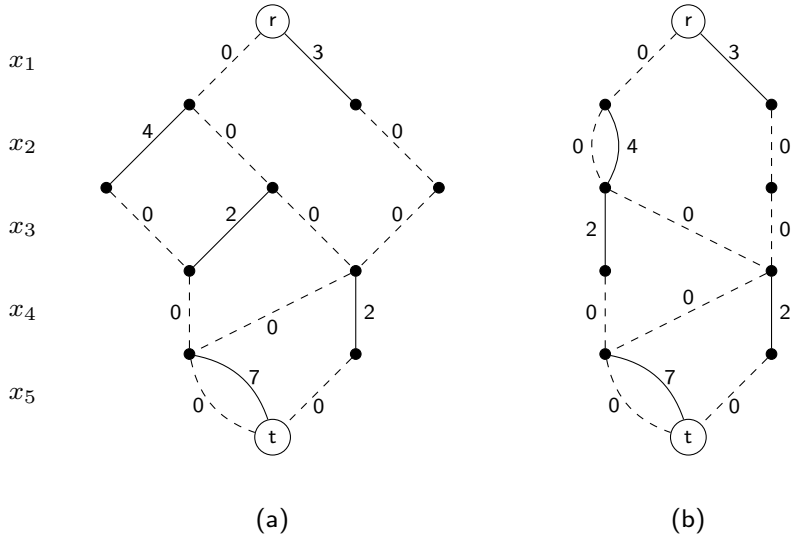


Figure 4.2: (a) Exact BDD and (b) relaxed BDD for the MISP on the graph in Figure 4.1.

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 $\bar{p}$, proving an upper bound of 13. □

Relaxed DDs were introduced by [5] 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 in [78, 74, 85] and [23]. Weighted DD relaxations were used to obtain optimization bounds in [26, 23], the former of which applied them to set covering and the latter to the maximum independent set problem.

The remainder of this chapter is organized as follows. In Section 4.2 we show how to modify the top-down compilation approach from Section 3.3 to generate relaxed DDs that observe an input-specified width. We show three modeling examples and provide a thorough computational analysis of relaxed DDs for the maximum independent set problem. In Section 4.3 we describe an alternative method to generate relaxed DDs, the *incremental refinement* procedure, and exemplify its application to a single machine makespan problem.

4.2 Construction by Top-Down Compilation

Relaxed DDs of limited width can be built by considering an additional step in the modeling framework for *exact DDs* described in Section 3.3. Recall that such framework relies on a DP model composed of a state space, transition functions, transition cost functions, and a root value. For relaxed DD, 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 (Rel-1) and (Rel-2), perhaps with an adjustment in the transition costs. The underlying goal of such rule is to create a relaxed DD that provides a tight bound given the maximum available width.

This rule is applied as follows. 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.

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 (Rel-1) and (Rel-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 4.2.4 we present a computational analysis of how the choice of nodes to merge influences the resulting optimization

Algorithm 3 Relaxed DD Top-Down Compilation for maximum width W

Create node $r = \hat{r}$ and let $L_1 = \{r\}$
for $j = 1$ to n **do**
  while $|L_j| > W$ **do**
    let $M = \text{node\_select}(L_j)$, $L_j \leftarrow (L_j \setminus M) \cup \{\oplus(M)\}$
    for all $u \in L_{j-1}$ and $i \in D_j$ with $b_i(u) \in M$ **do**
      $b_i(u) \leftarrow \oplus(M)$, $v\left(a_i(u)\right) \leftarrow \Gamma_M\left(v(a_i(u)), b_i(u)\right)$
  let $L_{j+1} = \emptyset$
  for all $u \in L_j$ and $d \in D_j$ **do**
    if $t_j(u, d) \neq \hat{0}$ **then**
      let $u' = t_j(u, d)$, add u' to L_{j+1}, and set $b_d(u) = u'$, $v(u, u') = h_j(u, u')$
Merge nodes in L_{n+1} into terminal node t

bound for the maximum independent set problem.

We remark that a research direction still unexplored is whether we could pick a *node_select* function that gives a formal guarantee on the resulting bound provided by a limited-width DD. We provide a simple example of such guarantee for the single machine makespan problem in Section 4.3.1.

4.2.1 Maximum Independent Set

The maximum independent set (MISP), first presented in Section 3.3.3, 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 3.3.3 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: $t_j(s^j, 0) = s^j \setminus \{j\}$, $\quad t_j(s^j, 1) = \begin{cases} s^j \setminus N(j) & , \text{ if } j \in s^j \\ \hat{0} & , \text{ if } j \notin s^j \end{cases}$
- cost functions: $h_j(s^j, 0) = 0$, $h_j(s^j, 1) = w_j$

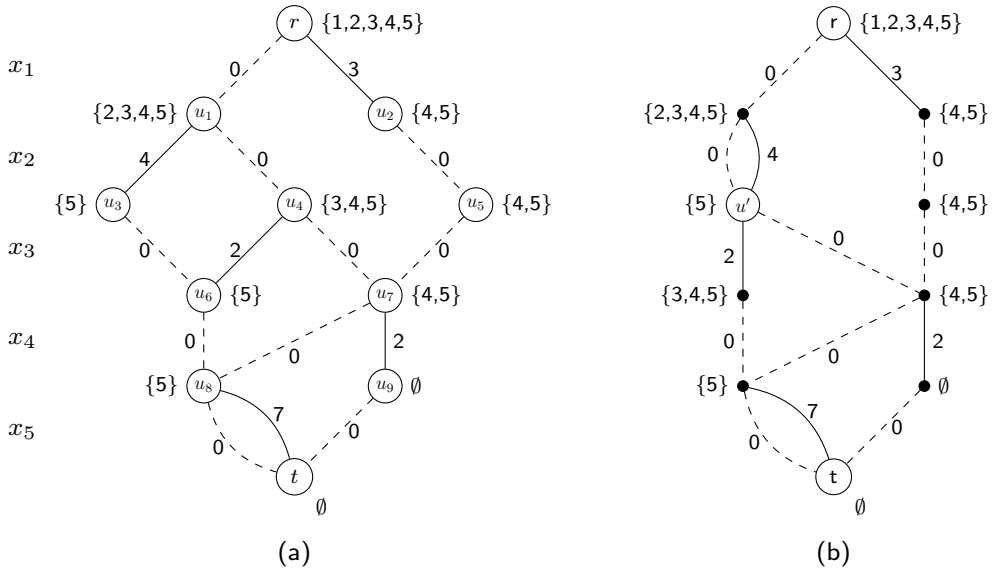


Figure 4.3: (a) Exact BDD with states for the MISP on the graph in Figure 4.1. (b) Relaxed BDD for the same problem instance.

- A root value of 0.

To create a relaxed DD for the MISP, we introduce a merging rule to 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.

Example 16 Figure 4.3(a) presents an exact DD for the MISP instance defined on the graph in Figure 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 $u' = u_2 \cup u_3 = \{3, 4, 5\}$, which reduces the width of the BDD to 2.

In the exact DD of Figure 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 Figure 4.3(b), the longest path corresponds to the solution $(0, 1, 1, 0, 1)$ and has length 13, which proves an upper bound of 13 on the objective function. Note that the longest path in the relaxed DD corresponds to an infeasible solution to that instance. □

4.2.2 Maximum Cut Problem

The maximum cut problem (MCP) was first presented in Section 3.3.7. Given a graph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$, 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 3.3.7 considers a state that represent 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: $S_k = \left\{ s^k \in \mathbb{R}^n \mid s^k_j = 0,\ j = 1, \ldots, k-1 \right\}$, 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

$$s^{k+1}_\ell = \left\{ \begin{array}{ll} s^k_\ell + w_{k\ell}, & \text{if } x_k = \text{S} \\ s^k_\ell - w_{k\ell}, & \text{if } x_k = \text{T} \end{array} \right\}, \quad \ell = k+1, \ldots, n$$

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

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

- root value: $v_r = \displaystyle\sum_{1 \le j < j' \le n} (w_{jj'})^-$

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 $u^{\text{new}}_j = \max\{u^1_j, u^2_j\}$ 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.

We therefore merge the nodes in M as follows. 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,

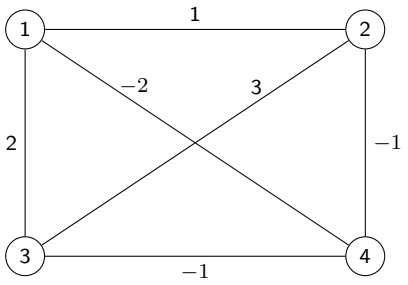


Figure 4.4: Graph with edge weights for the MCP .

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

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

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

Example 17 Figure 4.5 shows an exact DD and a relaxed DD for the MCP instance defined over the graph in Figure 4.4. The relaxed DD is obtained by merging nodes u_2 and u_3 during top-down construction. In the exact DD of Figure 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 Figure 4.5(b), the longest path corresponds to the solution $(\mathrm{S}, \mathrm{S}, \mathrm{S}, \mathrm{S})$ and has length 5, which proves 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 the following.

Lemma 14 *Let B be a relaxed BDD generated by Algorithm 1 for an instance $\mathcal{P}$ of the MCP. Suppose we add Δ to one state s^k_ℓ 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 1, the result is a relaxed BDD for $\mathcal{P}$.*

Proof. Let B' 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 $L_{k-1}, \ldots, L_n$

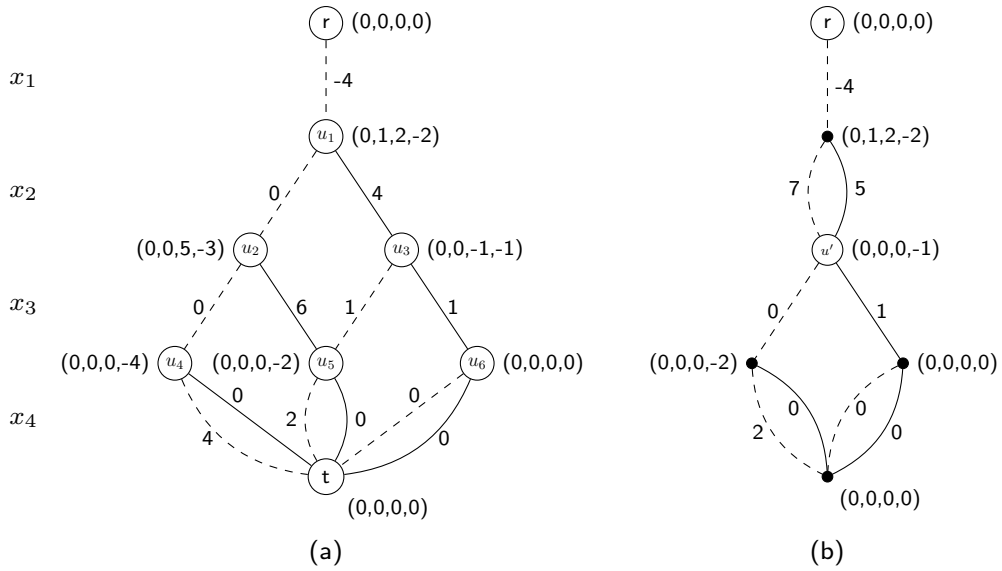


Figure 4.5: (a) Exact BDD with states for the MCP on the graph in Figure 4.4. (b) Relaxed BDD for the same problem instance.

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

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

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

THEOREM 15 *Operators $\oplus$ and Γ as defined in (MCP-relax) are valid relaxation operators for the MCP.*

Proof. We can achieve the effect of Algorithm 1 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. So we can let $\Delta = \pm\delta$ in Lemma 14 and conclude that each step of the procedure yields a relaxed BDD. □ ■

4.2.3 Maximum 2-Satisfiability Problem

The maximum 2-satisfiability problem was described in Section 3.3.8. 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 15.

THEOREM 16 *Operators $\oplus$ and Γ as defined in (MCP-relax) are valid relaxation operators for MAX-2SAT.*

4.2.4 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++.

Merging Heuristics

The selection of nodes to merge in a layer that exceeds the maximum allotted width W is critical for the construction of relaxation BDDs. Different selections may yield dramatic differences on 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 a stand-alone heuristic or combined with any of the following heuristics for the purpose of generating several relaxations.

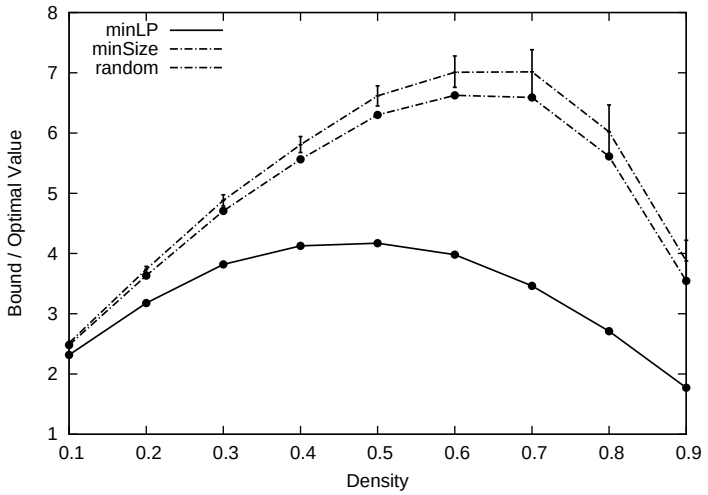


Figure 4.6: Bound quality vs. graph density for each merging heuristic, using the `random` instance set with `MPD` ordering and maximum BDD width 10. Each data point represents an average over 20 problem instances. The vertical line segments indicate the range obtained in 5 trials of the random heuristic.

- **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 2 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 tested the three merging heuristics on the `random` instance set. We set a maximum width of $W = 10$ and used variable ordering heuristic `MPD`. 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, because `random` yielded much weaker bounds than the other techniques tested. In light of these results, we use **minLP** as the merging heuristic for the remainder of the experiments.

Variable Ordering Heuristics

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. It is well known that finding orderings that minimize the size of BDDs (or even improving on a given ordering) is NP-hard [50, 30]. We found that the ordering of the vertices is the single most important parameter in creating small width exact BDDs and in proving tight bounds via relaxed BDDs.

Different orderings can yield exact BDDs with dramatically different widths, as discussed in details in Section 3.5 of this dissertation. In particular, recall Figure 3.16a from that section, which shows a path graph on 6 vertices with two different orderings given by $x_1, \ldots, x_6$ and $y_1, \ldots, y_6$. In Figure 3.16c we see that the vertex ordering $x_1, \ldots, x_6$ yields an exact BDD with width 2, while in Figure 3.16b 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 that they lie on the paths yields a BDD of width 1.

The orderings in Section 3.5 inspire variable ordering heuristics for generating relaxed BDD. We outline a few that are tested below. Note that the first two orderings are *dynamic*, in that we select the j-th vertex in the order based on the first $j-1$ vertices chosen and the partially constructed BDD. In contrast, the last ordering is *static*, in that the ordering is determined prior to building the BDD.

- **random**: Randomly select some vertex that has yet to be chosen. This may be used a stand-alone heuristic or combined with any of the following heuristics for the purpose of generating several relaxations.
- **minState**: Select the vertex v_j appearing in the fewest number of states in P. This minimizes the size of L_j, given the previous selection of vertices $v_1, \ldots, v_{j-1}$, since the only nodes in P that will appear in L_j are exactly those nodes containing v_j in their associated state. Doing so limits the number of merging operations that need to be performed.
- **MPD**: As proved in Section 3.5, a maximal path decomposition ordering of the vertices bounds the exact BDD width by the Fibonacci numbers, which grow slower than 2^j (the worst case). Hence this ordering limits the width of all layers, therefore limiting the number of merging operations necessary to build the BDD.

In first set of experiments, we wish to provide an empirical evidence that a variable ordering with a smaller exact BDD results in a relaxation BDD with a tighter bound. If this hypothesis holds, it would function as an additional motivation to study how variable orderings impact the width on problems with other combinatorial structures, as they may have a substantial effect on the optimization bound of relaxed DDs. The instances tested for this purpose were generated as follows. We first selected 5 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 each instance, which is approximately the largest graph size that the exact BDD can be built within our memory limits.

The tests are described next. For each instance and the three orderings random, minState, and MPD above, 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 variables orderings may yield BDDs with very different exact widths.

Figure 4.7 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 Figure 4.8 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 27684, 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 the previous set of experiments. A similar behavior is obtained if the same chart is plotted for other instances.

We now test the three variable ordering heuristics on the complete random instance set to analyze the quality of the bound. The results (Fig. 4.9) indicate that the MinState ordering is the best of the three. This is particularly true for sparse graphs, because the number of possible node states generated by dense graphs is relatively small. We therefore use MinState ordering for the remainder of the experiments.

Bounds vs. Maximum BDD Width

The purpose of this experiment is to analyze the impact of maximum BDD width on the resulting bound. Figure 4.10 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.

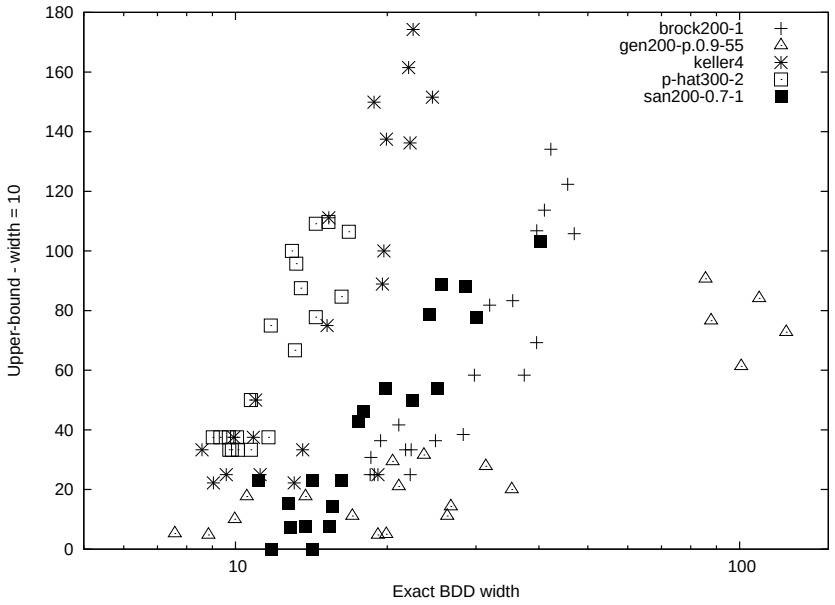


Figure 4.7: Bound of relaxation BDD vs. exact BDD width.

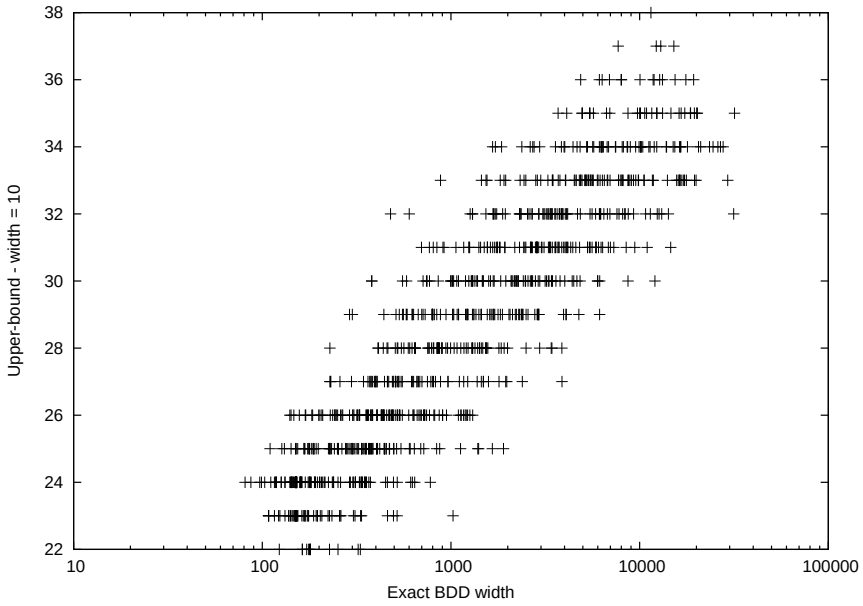


Figure 4.8: Bound of relaxation BDD vs. exact BDD width for `san200_0.7_1`.

Comparison with LP Relaxation

We now address the key question of how BDD bounds compare with bounds produced by a traditional LP relaxation and cutting planes. To obtain a tight initial LP relaxation, we used a *clique cover* model [70] of the maximum independent set problem, which requires computing a clique cover

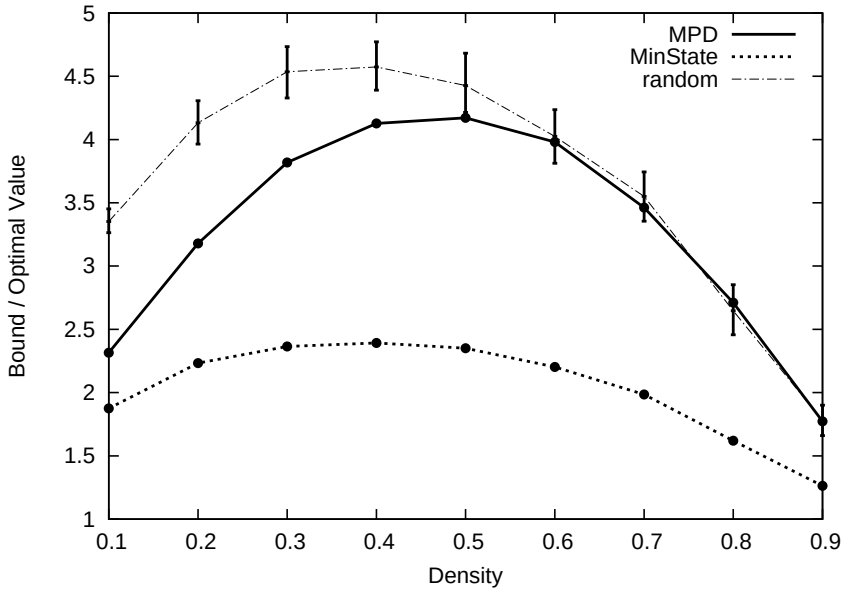


Figure 4.9: Bound quality vs. graph density for each variable ordering heuristic, using merge heuristic **minLP** and otherwise the same experimental setup as Fig. 4.6.

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

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

The clique cover $\mathcal{C}$ was computed using a greedy procedure as follows. Starting with $\mathcal{C} = \emptyset$, 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 $\mathcal{C}$, 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 MIP 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 **random** instances appear in Table 4.1 and are plotted in Fig. 4.11. The table

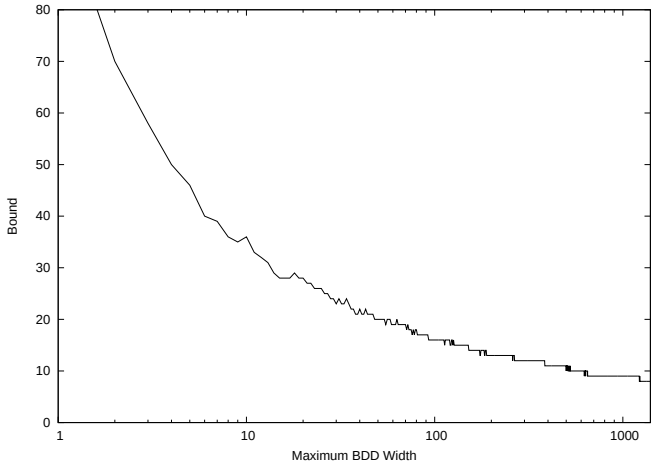


Figure 4.10: Relaxation bound vs. maximum BDD width for **dimacs** instance `p-hat_300-1`.

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 scatterplot in Fig. 4.13 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.

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. Increasing the width to 10,000 yields bounds that are superior for every instance, as revealed by the scatter plot in Fig. 4.15. The time required is about a third as much as LP overall, but somewhat more for sparse instances.

The results for **dimacs** instances appear in Table 4.2 and Fig. 4.12, with scatter plots in Figs. 4.16–4.18. 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

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

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 10000. Each graph density setting is represented by 20 problem instances.

	Bound quality (geometric mean)					Time in seconds (shifted geometric mean)				
	LP relaxation		*BDD relaxation*			*LP relaxation*		*BDD relaxation*		
Density	LP only	LP+cuts	100	1000	10000	LP only	LP+cuts	100	1000	10000
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**

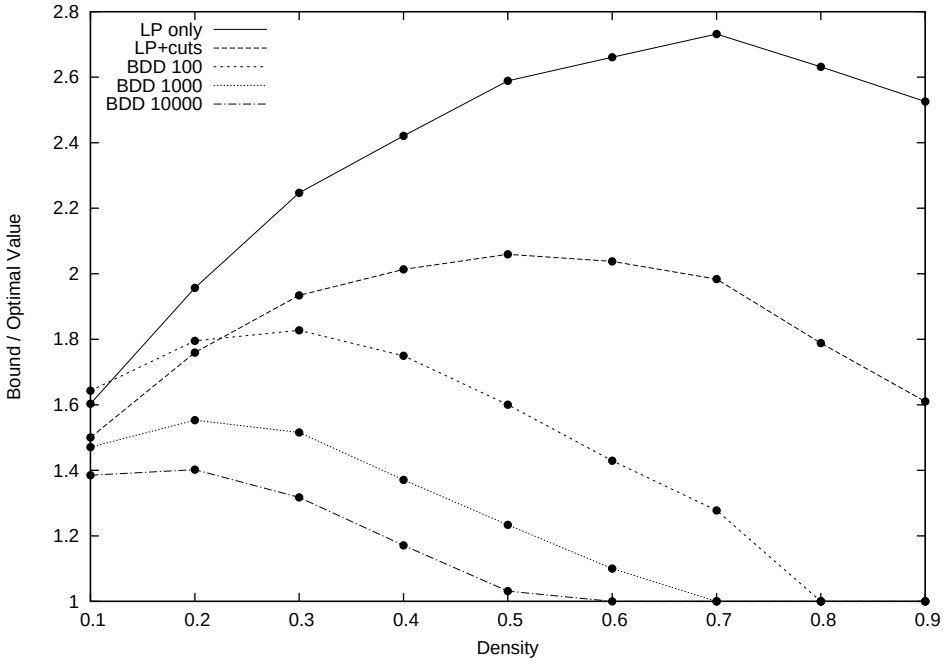


Figure 4.11: Bound quality vs. graph density for **random** instances, showing results for LP only, LP plus cutting planes, and BDDs with maximum width 100, 1000, and 10000. Each data point is the geometric mean of 20 instances.

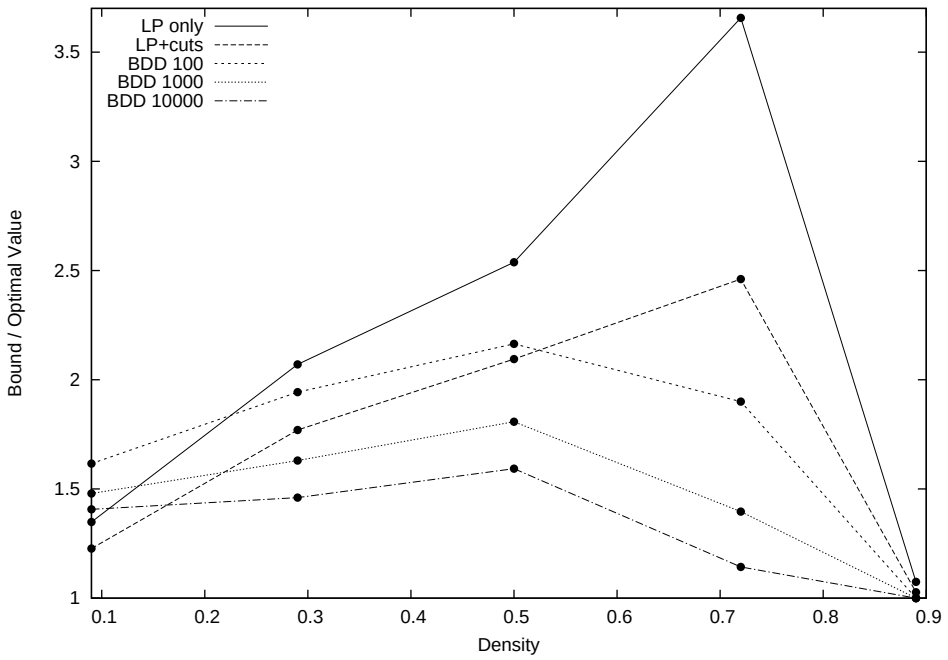


Figure 4.12: Bound quality vs. graph density for **dimacs** instances, showing results for LP only, LP plus cutting planes, and BDDs with maximum width 100, 1000, and 10000. Each data point is the geometric mean of instances in a density interval of width 0.2.

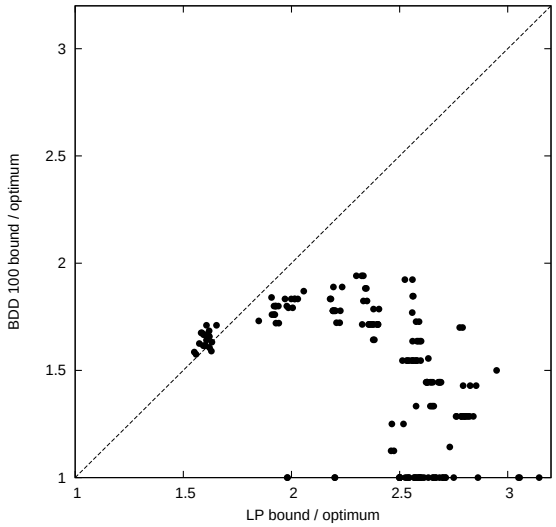


Figure 4.13: Bound quality for an LP relaxation (no cuts) vs. width 100 BDDs for **random** instances. Each data point represents one instance. The time required is about the same overall for the two types of bounds.

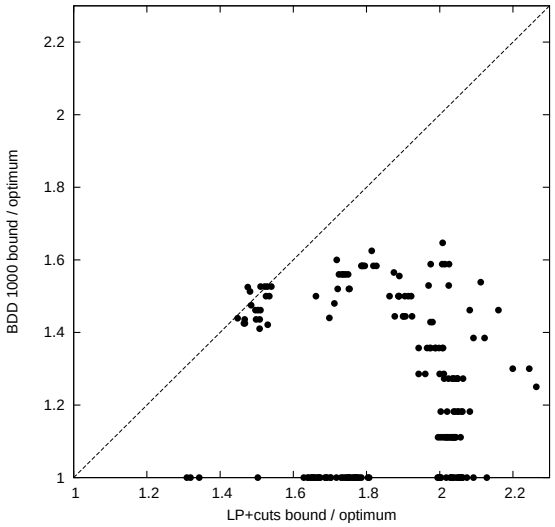


Figure 4.14: Bound quality for an LP relaxation with cuts vs. width 1000 BDDs for **random** instances. The BDD bounds are obtained in about 5% of the time required for the LP bounds.

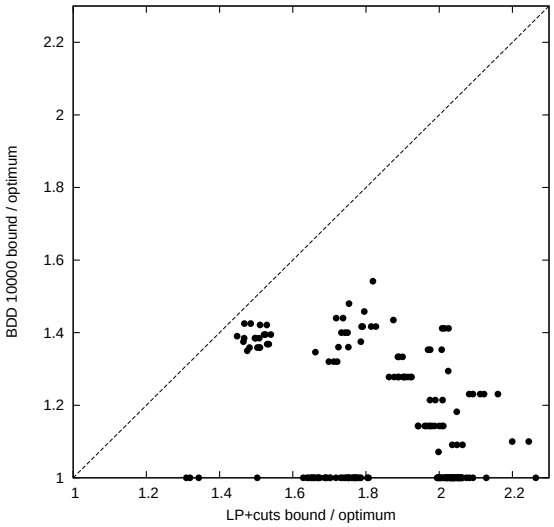


Figure 4.15: Bound quality for an LP relaxation with cuts vs. width 10000 BDDs for **random** instances. The BDD bounds are obtained in less time overall than the LP bounds, but somewhat more time for sparse instances.

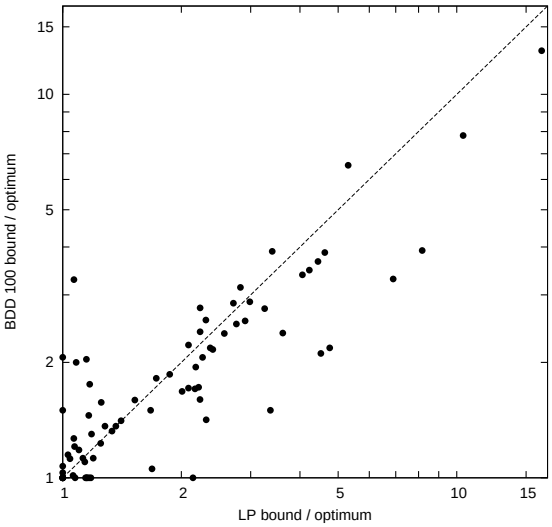


Figure 4.16: Bound quality for an LP relaxation (no cuts) vs. width 100 BDDs for **dimacs** instances. The BDD bounds are obtained in generally less time for all but the sparsest instances.

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 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.

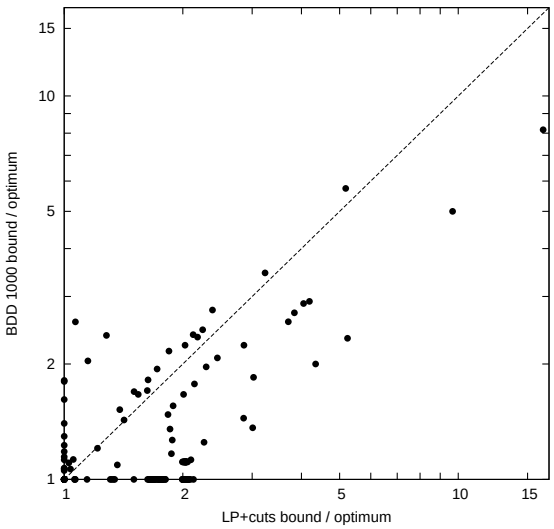


Figure 4.17: Bound quality for an LP relaxation with cuts vs. width 1000 BDDs for **dimacs** instances. The BDD bounds are obtained in about 15% as much time overall as the LP bounds.

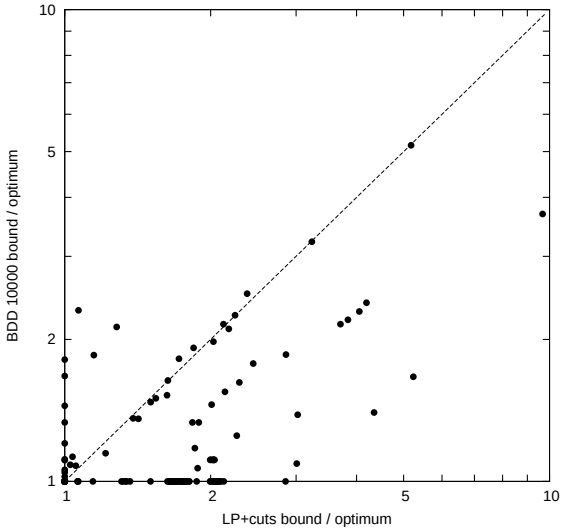


Figure 4.18: Bound quality for an LP relaxation with cuts vs. width 10000 BDDs for **dimacs** instances. The BDD bounds are generally obtained in less time for all but the sparsest instances.

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, 1000, and 10000.

		Bound quality (geometric mean)					Time in seconds (shifted geometric mean)				
Avg.		*LP relaxation*		*BDD relaxation*			*LP relaxation*		*BDD relaxation*		
Density	Count	LP only	LP+cuts	100	1000	10000	LP only	LP+cuts	100	1000	10000
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**

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 1000		Width 10000	
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

4.3 Construction by Separation: Filtering and Refinement

An alternative procedure to compile relaxed DDs can be obtained by modifying the separation procedure from Section 3.4 in a straightforward way. Recall that such 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 by [78] and [85] 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 associate 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 4.3.1.

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 using an 1-width DD as depicted in Figure 3.12a. 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

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

Let $B' = (U', A')$ be a DD such that $\text{Sol}(B') \supseteq \text{Sol}(\mathcal{P})$.
while $\exists$ constraint C violated by B' **do**
  Let $s(u) = \chi$ for all nodes $u \in B'$
  $s(r) := \hat{r}$
  for $j = 1$ to n **do**
    // Filtering
    for $u \in L_j$ **do**
      for each arc $a = (u, v)$ leaving node u **do**
        if $t_j^C(s(u), d(a)) \neq \hat{0}$ **then**
          Remove arc a from B
    // Refinement
    for $u \in L_j$ **do**
      for each arc $a = (u, v)$ leaving node u **do**
        if $s(v) = \chi$ **then**
          $s(v) = t_j^C(s(u), d(a))$
        else if $s(v) \neq t_j^C(s(u), d(a))$ **and** $|L_j| < W$ **then**
          Remove arc (u, v)
          Create new node v' with $s(v') = t_j^C(u, d(a))$
          Add arc (u, v')
          Copy outgoing arcs from v as outgoing arcs from v'
          $L_j := L_j \cup \{v'\}$
        else
          Update $s(v)$ with $t_j^C(s(u), d(a))$.

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 $\hat{0}$ 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.

4.3.1 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 3.3.6. Given a positive integer n, let $\mathcal{J} = \{1, \ldots, n\}$ 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 from 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 3.3.6, the MMP can be written as

the following optimization problem:

$$\begin{aligned} \min \ & \sum_{i=1}^{n} p_{i,x_i} \\ & \textsc{Alldifferent}(x_1, \ldots, x_n) \\ & x_i \in \{1, \ldots, n\}, \ \ i = 1, \ldots, n \end{aligned} \tag{4.1}$$

We will now show how to define the filtering and refinement operations for the constraint (4.1). The feasible solutions are defined by all vectors x that satisfy the ALLDIFFERENT constraint in (4.1); that is, they are the permutations of J without repetition. The states used for filtering and refinement for the ALLDIFFERENT were initially introduced by [5] and [85].

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 condition holds. We can directly use the state and transition function defined in Section 3.3.6; 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 $All_u^{\downarrow} \subseteq \mathcal{J}$ and $Some_u^{\downarrow} \subseteq \mathcal{J}$ 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 3.3.6), 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

$$All_v^{\downarrow} = \bigcap_{a=(u,v) \in in(v)} (All_u^{\downarrow} \cup \{d(a)\}), \tag{4.2}$$

$$Some_v^{\downarrow} = \bigcup_{a=(u,v) \in in(v)} (Some_u^{\downarrow} \cup \{d(a)\}). \tag{4.3}$$

For example, in Figure 4.19b we have $All_{v_1}^{\downarrow} = \{1\}$ and $Some_u^{\downarrow} = \{1, 2, 3\}$.

Lemma 17 *An arc $a = (u, v)$ is infeasible if any of the following conditions holds:*

$$d(a) \in All_u^{\downarrow}, \tag{4.4}$$

$$|Some_u^{\downarrow}| = \ell(a) \quad \textit{and} \quad d(a) \in Some_u^{\downarrow}. \tag{4.5}$$

Proof. The proof argument follows from [5]. Let π' be any partial ordering identified by a path from $\mathbf{r}$ to u that does not assign any job more than once. In condition (4.4), $d(a) \in All_u^{\downarrow}$ 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 (4.5), notice first that the paths from $\mathbf{r}$ 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 (4.4) and (4.5) can be applied in lines 6 to 10 in Algorithm 4 to remove infeasible arcs. For example, in Figure 4.19b the two shaded arcs are infeasible. The arc (u_1, v_1) with label 1 is infeasible due to condition (4.4) since $All_{u_1}^{\downarrow} = \{1\}$. The arc (v_A, t) with label 2 is infeasible due to condition (4.5) 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 [85], we define two new states $All_u^{\uparrow} \subseteq \mathcal{J}$ and $Some_u^{\uparrow} \subseteq \mathcal{J}$ for each node u of $\mathcal{M}$. They are equivalent to the states $All_u^{\downarrow}$ and $Some_u^{\downarrow}$, but now they are computed with respect to the paths from $\mathbf{t}$ to u instead of the paths from $\mathbf{r}$ to u. As before, they are recursively obtained through the relations

$$All_u^{\uparrow} = \bigcap_{a=(u,v) \in out(u)} (All_v^{\uparrow} \cup \{d(a)\}), \tag{4.6}$$

$$Some_u^{\uparrow} = \bigcup_{a=(u,v) \in out(u)} (Some_v^{\uparrow} \cup \{d(a)\}), \tag{4.7}$$

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

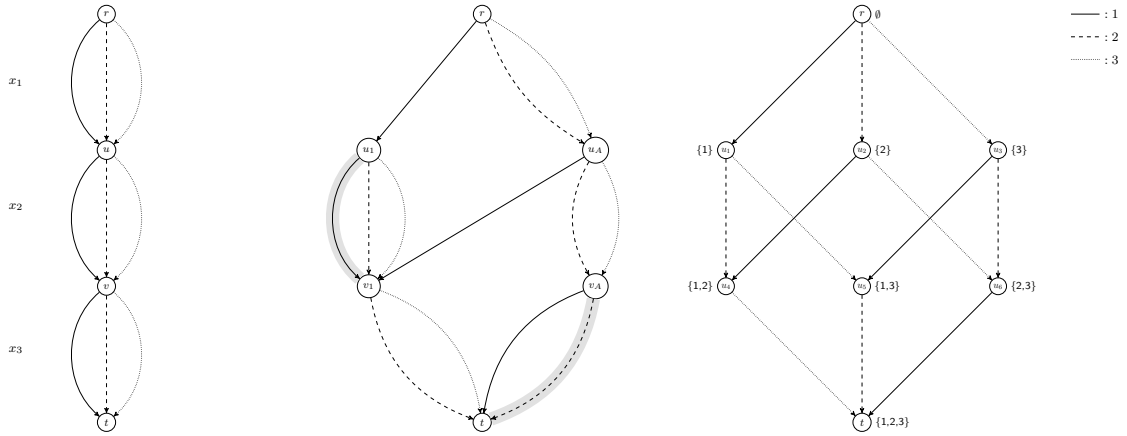
Lemma 18 *An arc $a = (u, v)$ is infeasible if any of the following conditions holds:*

$$d(a) \in All_v^{\uparrow}, \tag{4.8}$$

$$|Some_v^{\uparrow}| = n - \ell(a) \quad \textit{and} \quad d(a) \in Some_v^{\uparrow}, \tag{4.9}$$

$$|Some_u^{\downarrow} \cup \{d(a)\} \cup Some_v^{\uparrow}| < n. \tag{4.10}$$

Proof. The proofs for conditions (4.8) and (4.9) follow from an argument in [85] and are analogous to the proof of Lemma 17. Condition (4.10) implies that any ordering identified by a path containing a will never assign all jobs $\mathcal{J}$. ■



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

Figure 4.19: Three phases of refinement, as described in Example 18.

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 7.

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

THEOREM 19 *Let $W > 0$ be the maximum MDD width. There exists a relaxed MDD $\mathcal{M}$ where at least $\lfloor \log_2 W \rfloor$ jobs are assigned to exactly one position in all orderings identified by $\mathcal{M}$.*

Let us represent the job priorities by defining a *ranking* of jobs $\mathcal{J}^* = \{j_1^*, \ldots, j_n^*\}$, where jobs with smaller index in $\mathcal{J}^*$ have a higher priority. We can thus achieve the desired property of our heuristic refinement by constructing the relaxed MDD $\mathcal{M}$ based on Theorem 19, where we ensure that the jobs exactly represented in $\mathcal{M}$ are those with a higher ranking.

Before proving Theorem 19, we first identify conditions on when a node violates the desired refinement property and needs to be modified. To this end, let $\mathcal{M}$ be any relaxed MDD. Assume the states $All_u^{\downarrow}$ and $Some_u^{\downarrow}$ as described before are computed for all nodes u in $\mathcal{M}$, and no arcs satisfy the infeasibility conditions (4.4) to (4.10). We have the following Lemma.

Lemma 20 *A job j is assigned to exactly one position in all orderings identified by $\mathcal{M}$ if and only if $j \notin Some_u^{\downarrow} \setminus All_u^{\downarrow}$ for all nodes $u \in \mathcal{M}$.*

Proof. Suppose first that a job j is assigned to exactly one position in all orderings identified by $\mathcal{M}$, and take a node u in $\mathcal{M}$ 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 $\mathcal{M}$. 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 (4.4). Thus, no job is assigned more than once in any ordering encoded by $\mathcal{M}$. Finally, rule (4.10) ensures that j is assigned exactly once in all paths. ■ ■

We now provide a constructive proof for Theorem 19.

Proof. Proof of Theorem 19 Let $\mathcal{M}$ be an 1-width relaxation. We can obtain the desired MDD applying filtering and refinement on $\mathcal{M}$ in a top-down approach as described in Section 4.3. For filtering, remove all arcs satisfying the infeasibility rules (4.4) and (4.5). 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 the conditions of Lemma 20 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 20 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, $j \notin Some_v^{\downarrow}$, hence $j \notin Some_{u_2}^{\downarrow} \setminus All_{u_2}^{\downarrow}$ by relation (4.2). Analogously, we can show $j \in All_{u_1}^{\downarrow}$, thus $j \notin Some_{u_1}^{\downarrow} \setminus All_{u_1}^{\downarrow}$.

Since the jobs $\mathcal{J}$ 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 20 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 $\mathcal{M}$. ■ ■

We can utilize Theorem 19 to guide our top-down approach for filtering and refinement, following the refinement heuristic based on the job ranking $\mathcal{J}^*$ described in the proof of Theorem 19. 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 $\mathcal{J}^*$, identify the nodes u such that $j^* \in Some_u^{\downarrow} \setminus All_u^{\downarrow}$ and split them into two nodes u_1 and u_2, where an incoming arc $a = (v, u)$ is redirected to u_1 if $j^* \in (All_v^{\downarrow} \cup \{d(a)\})$ 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 $\mathcal{M}$ for certain structured problems, which we will show in Section 7.7. 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 $\mathcal{M}$ is still a valid relaxation and may provide a stronger representation.

Example 18 Let $\mathcal{J} = \{1, 2, 3\}$ and assume jobs are ranked lexicographically. Given the relaxed DD in Figure 4.19a, Figure 4.19b 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 4.19c 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 an 1-width relaxation and the constraint set is composed of only one Alldifferent). □

Chapter 5

Restricted Decision Diagrams

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 is 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 [34] and the maximum clique problem [69, 112]. In contrast, general-purpose heuristics have received much less attention in the literature. The vast majority of the general techniques are embodied in integer programming technology, such as the *feasibility pump* [57] and the *pivot, cut, and dive* heuristic [52]. A survey of heuristics for integer programming is presented by [65, 66] and [27]. Local search methods for binary problems can also be found in [1] and [28].

We introduce a new general-purpose method for obtaining a set of feasible solutions for discrete optimization problems. Our method is based on an under-approximation of the feasible solution set using *restricted* DDs. Restricted DDs can be perceived as a counterpart of the concept of relaxed DD introduced in Chapter 4.

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

$$\text{Sol}(\mathcal{P}) \;\supseteq\; \text{Sol}(B) \tag{Res-1}$$

$$f(x^p) \;\geq\; v(p), \text{ for all } r\text{–}t \text{ paths } p \text{ in } B \text{ for which } x^p \in \text{Sol}(\mathcal{P}) \tag{Res-2}$$

Suppose $\mathcal{P}$ is a maximization problem. In Chapter 3, we showed that an exact DD reduces

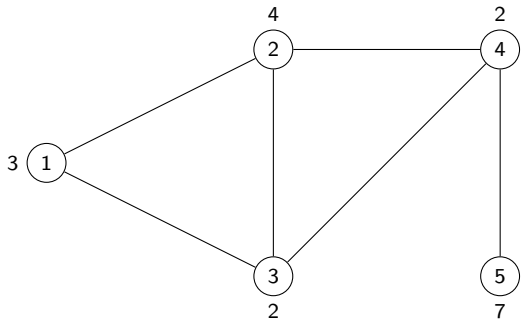


Figure 5.1: Graph with vertex weights for the MISP.

discrete optimization to a longest-path problem: If p is a longest path in a BDD B that is exact for $\mathcal{P}$, then x^p is an optimal solution of $\mathcal{P}$, and its length $v(p)$ is the optimal value $z^*(\mathcal{P}) = f(x^p)$ of $\mathcal{P}$. When B is restricted for $\mathcal{P}$, a longest path p provides a *lower bound* on the optimal value. The corresponding solution x^p is *always* feasible and $v(p) \leq z^*(\mathcal{P})$. 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.

Example 19 Consider the graph and vertex weights depicted in Figure 5.1 (the same from Figure 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 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 proves a lower bound of 10 on the objective function. □

The remainder of this chapter is divided into two sections. In Section 5.2 we show how to modify the top-down compilation approach from Section 3.3 to generate restricted DDs that observe an input-specified width. In Section 5.3 we present a thorough computational study of the bound provided by restricted DD, particularly focusing on the set covering and set packing problems.

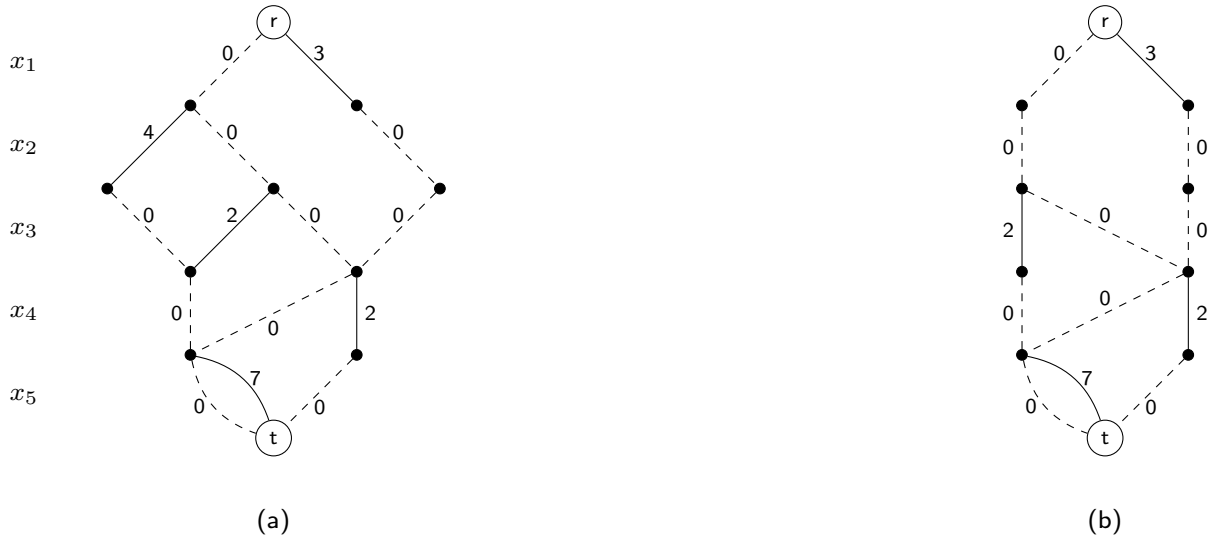


Figure 5.2: (a) Exact BDD and (b) restricted BDD for the MISP on the graph in Figure 5.1.

5.2 Construction by Top-Down Compilation

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 $\mathcal{P}$ and a maximum width W, Algorithm 5 outputs a restricted DD for $\mathcal{P}$. Note that it is similar to Algorithm 1 except for lines 3 to 5. Condition (Res-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 (Res-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 DD. To this end, we modifying the operator $\oplus(M)$ so that the Algorithm outputs restrictions instead of relaxations. For example, in the MISP relaxation described in Section 4, we could apply the *intersection* operator instead of the union. Such technique will not be exploited in this dissertation, but it could be useful to ensure certain properties of the restricted DD (e.g., it can be show 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 perform 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 in Sections 3.3.4 and 3.3.5, respectively. We evaluate our method by comparing the

Algorithm 5 Restricted DD Top-Down Compilation for maximum width W

Create node $r = \hat{r}$ and let $L_1 = \{r\}$
for $j = 1$ to n **do**
  while $|L_j| > W$ **do**
    let $M = \text{node\_select}(L_j)$
    $L_j \leftarrow (L_j \setminus M)$
  let $L_{j+1} = \emptyset$
  for all $u \in L_j$ and $d \in D_j$ **do**
    if $t_j(u, d) \neq \hat{0}$ **then**
      let $u' = t_j(u, d)$, add u' to L_{j+1}, and set $b_d(u) = u'$, $v(u, u') = h_j(u, u')$
Merge nodes in L_{n+1} into terminal node t

bounds provided by a restricted BDD with the ones obtained via state-of-the-art integer programming technology (IP). 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 [28] 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 3.3.4 and 3.3.5. 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 band-

width 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 ([102, 44, 54, 72, 103, 110, 121]). 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 [61] and IP finds 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 containing 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 Relation between Solution Quality and Maximum BDD Width

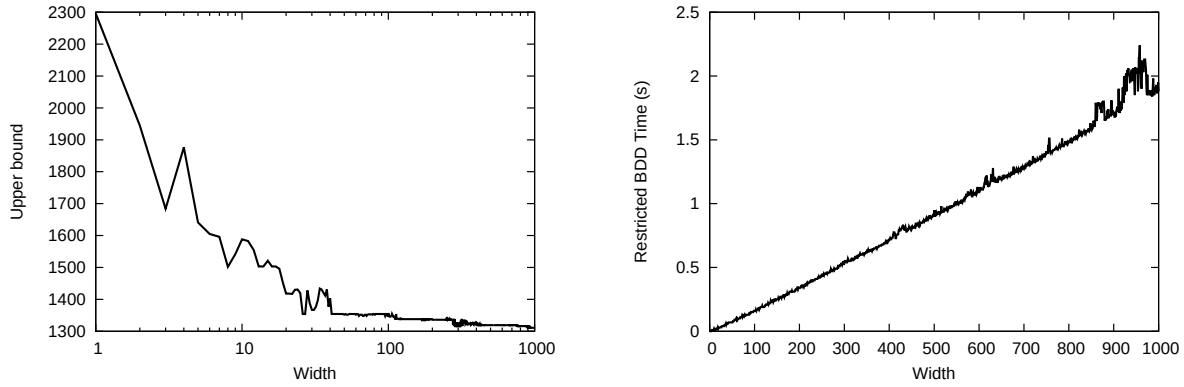
We first analyze the impact of the maximum width W on the solution quality provided by a restricted BDD. To this end, we report the generated bound versus maximum width W obtained for a set covering instance with $n = 1000$, $k = 100$, $b_w = 140$, and a cost vector c where each c_j was chosen uniformly 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 5.3a depicts the resulting bounds, where the width axis is in log-scale, and Figure 5.3b presents the total time to generate the W-restricted BDD and extract its best solution. We tested all W in the set $\{1, 2, 3, \ldots, 1000\}$. 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.

5.3.3 Set Covering

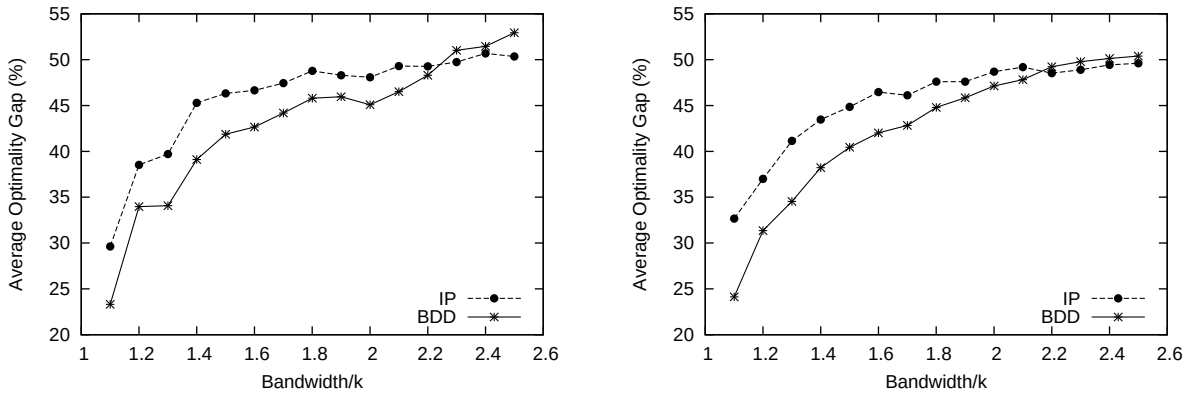
First, we report the results for two representative classes of instances for the set covering problem. In the first class, we studied the effect of b_w on the quality of the bound. To this end, we fixed $n = 500$, $k = 75$, and considered b_w as a multiple of k, namely $b_w \in \{\lfloor 1.1k \rfloor, \lfloor 1.2k \rfloor, \ldots, \lfloor 2.6k \rfloor\}$. 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 $n = 500$, $k \in \{25, 50, \ldots, 250\}$, 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



(a) Upper bound. (b) Time.

Figure 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.



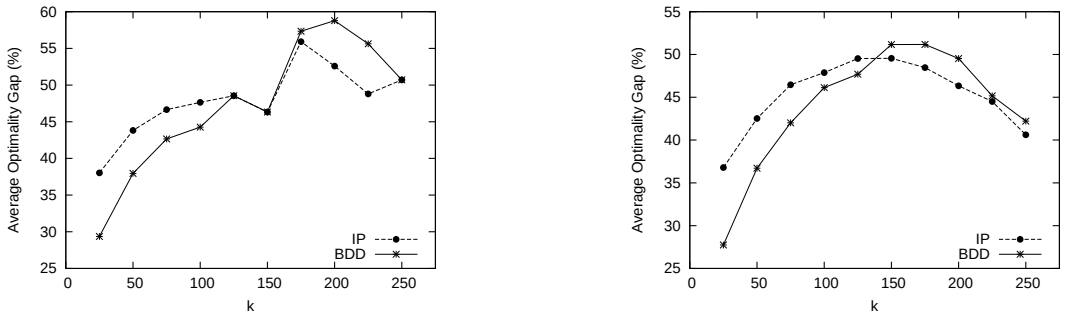
(a) Combinatorial. (b) Weighted.

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

second cost function, let nc_j be the number of constraints that include variable x_j, $j = 1, \ldots, n$. 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 Figure 5.4. Each data point in the figure represents the average optimality gap, over the instances with that configuration. We observe that



(a) Combinatorial.

(b) Weighted.

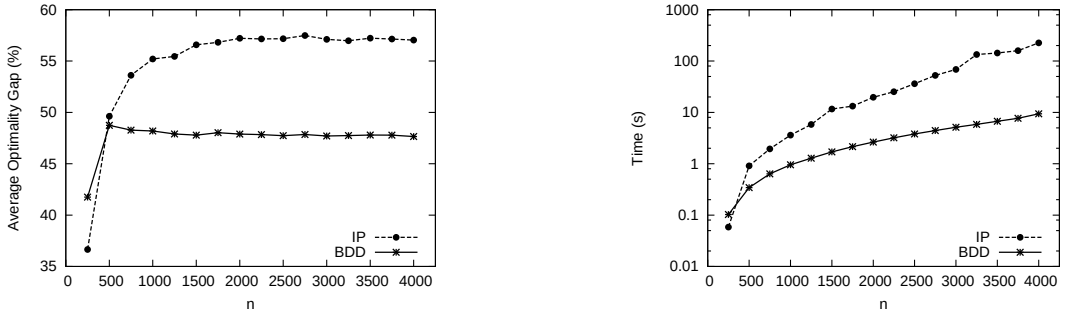
Figure 5.5: Average optimality gaps for combinatorial and weighted set covering instances with $n = 500$, varying k, and $b_w = 1.6k$.

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 to 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 we notice that the gap provided by the restricted BDD is generally better in comparison to 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 Figure 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 decrease 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 $n \in \{250, 500, 750, \ldots, 4000\}$, $k = 75$, and $b_w = 2.2k = 165$ to this end. The choice of k and b_w was motivated by Figure 5.4b, 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 Figures 5.6a and 5.6b, respectively. The y axis in Figure 5.6b is in logarithm 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 constants. However, IP times grow in a much faster rate than restricted BDD times. In particular, with



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

Figure 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 in logarithm scale.

$n = 4000$, the BDD times are approximately two orders-of-magnitude faster than the corresponding IP times.

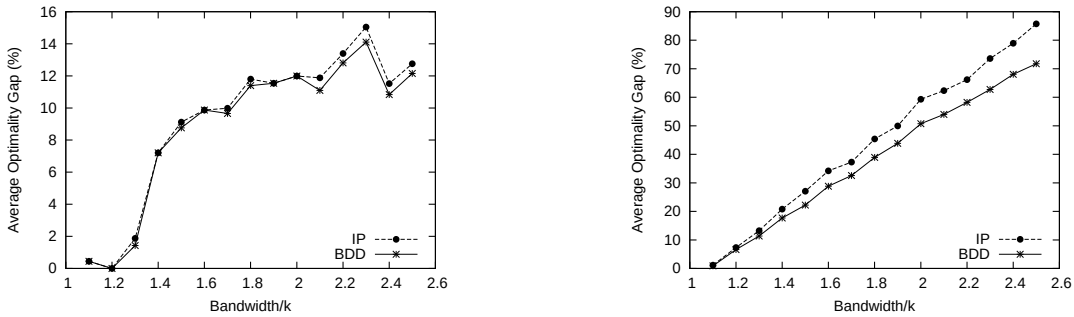
5.3.4 Set Packing

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

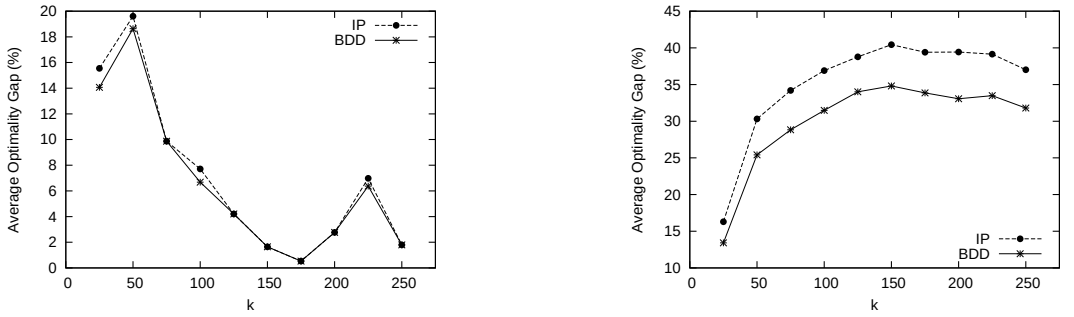
Similarly to the set covering problem, experiments were performed with two types of objective function coefficients. The first, $c = \mathbf{1}$, yields the *combinatorial* set packing problem. For the second cost function, let nc_j again denote the number of constraints that include variable x_j, $j = 1, \ldots, n$. 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 Figure 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



(a) Combinatorial. (b) Weighted.

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



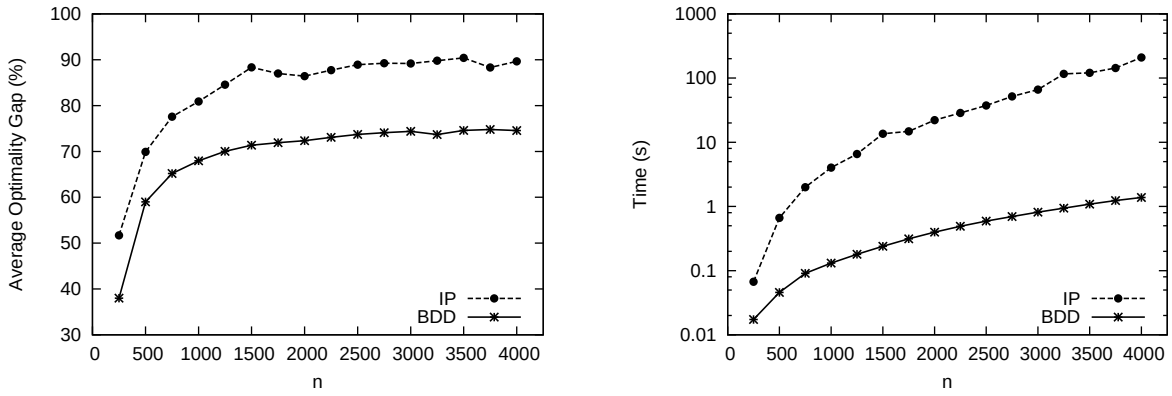
(a) Combinatorial. (b) Weighted.

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

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

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



(a) Average Optimality Gap (in %).

(b) Time (in seconds).

Figure 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 in logarithm scale.

The average optimality gap and solving times are presented in Figures 5.9a and 5.9b, 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.

Chapter 6

Branch-and-bound Based on Decision Diagrams

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.

We propose an alternative branch-and-bound method in which decision diagrams take over the functions of the traditional LP relaxation. As we analyzed in Chapters 4 and 5, limited-size 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 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 to a leading commercial IP solver in Section 6.2.3. 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 to 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 speed-ups.

This chapter is organized as follows. The branch-and-bound algorithm is presented in Section 6.2, where we demonstrate how to branch on nodes and provide a computational study on three classical combinatorial problems. The parallel branch-and-bound is described in Section 6.3. In this case, we will particularly focus on the maximum independent set problem to illustrate the related concepts and perform an experimental analysis.

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 DD 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 MDDs.

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, $B_{rt} = B$.

6.2.1 Exact Cutsets

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

As an illustration, Figure 6.1(a) duplicates the relaxed BDD $\bar{B}$ from Figure 3.4 and labels the nodes labeled exact (E) or relaxed (R). Node $\bar{u}_4$ in $\bar{B}$ is an exact node because all incoming paths (there is only one) lead to the same state $\{4, 5\}$. Node $\bar{u}_3$ 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 $\bar{u}_1$ an $\bar{u}_4$ form one possible exact cutset of $\bar{B}$.

We now show that an exact cutset provides an exhaustive enumeration of subproblems. If B is an exact BDD for binary optimization problem $\mathcal{P}$, and let $v^*(B_{uu'})$ be the length of a longest u–u' path in $B_{uu'}$. For a node u in B, we define $\mathcal{P}|_u$ to be the restriction of $\mathcal{P}$ whose feasible solutions

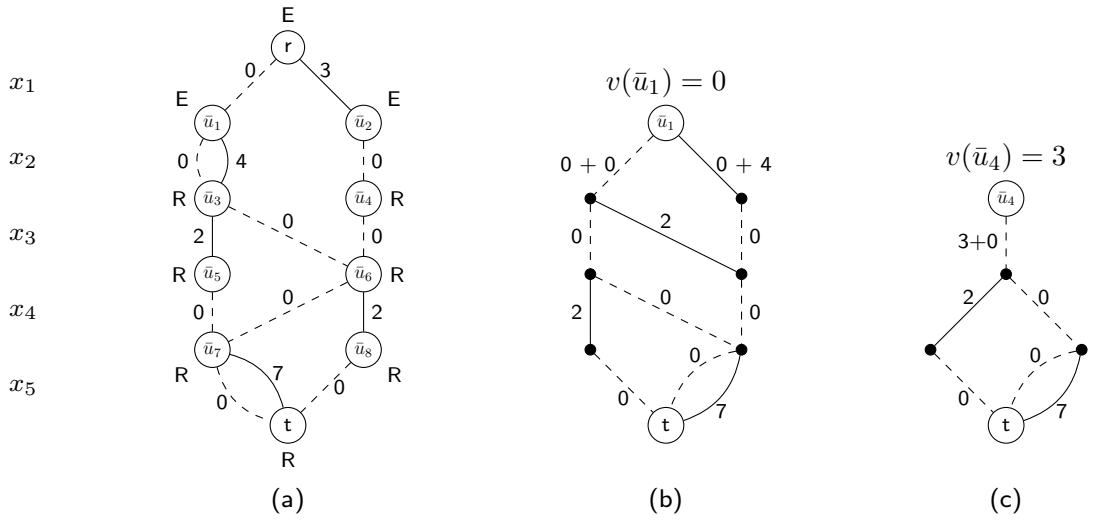


Figure 6.1: (a) Relaxed BDD for the MISP on the graph in Figure 3.3 with nodes labeled as exact (E) or relaxed (R); (b) exact BDD for subproblem corresponding to $\bar{u}_1$; (c) exact BDD for subproblem corresponding to $\bar{u}_4$.

correspond to r–t paths of B that contain u. Recall that $z^*(\mathcal{P})$ is the optimal value of $\mathcal{P}$.

Lemma 21 *If B is an exact BDD for $\mathcal{P}$, then for any node u in B,*

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

Proof. Proof $z^*(\mathcal{P}|_u)$ 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 22 *Let $\bar{B}$ be a relaxed BDD created by Algorithm 1 using a valid DP model of binary optimization problem $\mathcal{P}$, and let S be an exact cutset of $\bar{B}$. Then*

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

Proof. Proof Let B be the exact BDD for $\mathcal{P}$ 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

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

where the second equation is due to Lemma 21. □ ■

Algorithm 6 Branch-and-Bound Algorithm

initialize $Q = \{r\}$, where r is the initial DP state
let $z_{\text{opt}} = -\infty$, $v^*(r) = 0$
while $Q \neq \emptyset$ **do**
  $u \leftarrow$ select_node(Q), $Q \leftarrow Q \setminus \{u\}$
  create restricted BDD B'_{ut} using Algorithm 1 with root u and $v_r = v^*(u)$
  if $v^*(B_{ut}) > z_{\text{opt}}$ **then**
    $z_{\text{opt}} \leftarrow v^*(B')$
  if B'_{ut} is not exact **then**
    create relaxed BDD $\bar{B}_{ut}$ using Algorithm 1 with root u and $v_r = v^*(u)$
    if $v^*(\bar{B}_{ut}) > z_{\text{opt}}$ **then**
      let S be an exact cutset of $\bar{B}_{ut}$
      for all $u' \in S$ **do**
        let $v^*(u') = v^*(u) + v^*(\bar{B}_{uu'})$, add u' to Q
return z_{opt}

6.2.2 Enumeration of Subproblems

We solve a binary optimization problem $\mathcal{P}$ by a branching procedure in which we enumerate a set of subproblems $\mathcal{P}|_u$ 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 $z^*(\mathcal{P}|_u)$ 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 $z^*(\mathcal{P}|_u)$ 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, while open nodes remain, we select a node u from Q. We first obtain a lower bound on $z^*(\mathcal{P}|_u)$ 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 $z^*(\mathcal{P}|_u)$ by building a relaxed BDD $\bar{B}_{ut}$ as described above. If we cannot prune the search using this bound, we identify an exact cutset S of $\bar{B}_{ut}$ and add the nodes in S to Q. Because S is exact, for each $u' \in S$ we know that $v^*(u') = v^*(u) + v^*(\bar{B}_{uu'})$. The search terminates when Q is empty, at which point the incumbent solution is optimal by Theorem 22.

As an example, consider again the relaxed BDD $\bar{B}$ in Figure 6.1(a). The longest path length in

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

Exact Cutset Selection

Given a relaxed BDD, there are many exact cutsets. Here we present three such cutsets and experimentally evaluate them in Section 6.2.3.

- *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 $\bar{B}$, define the *last exact layer* of $\bar{B}$ to be the set of nodes $\text{LEL}(\bar{B}) = L_{j'}$, where j' is the maximum value of j for which each node in L_j is exact. In the relaxed BDD $\bar{B}$ of Figure 6.1(a), $\text{LEL}(\bar{B}) = \{\bar{u}_1, \bar{u}_2\}$.

- *Frontier cutset (FC).* For a relaxed BDD $\bar{B}$, define the *frontier cutset* of B to be the set of nodes

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

  In the example of Figure 6.1(a), $\text{FC}(\bar{B}) = \{\bar{u}_1, \bar{u}_4\}$. A frontier cutset is an exact cutset, due to the following.

  Lemma 23 *If $\bar{B}$ is a relaxed BDD that is not exact, then $\text{FC}(\bar{B})$ is an exact cutset.*

  Proof. Proof By the definition of a frontier cutset, each node in the cutset is exact. We need only show that each solution $x \in \text{Sol}(\bar{B})$ contains some node in $\text{FC}(\bar{B})$. But the path p corresponding to x ends at t, which is relaxed because $\bar{B}$ 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 $\text{FC}(\bar{B})$, as desired. □ ■

6.2.3 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 the most 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 8GB RAM. The BDD-based algorithm was implemented in C++. The commercial IP solver 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.

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 $\text{rank}(u) = v^*(u)$. We used the variable ordering heuristic in Section **TODO**: 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. We used FC cutsets for all MISP tests.

For 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 pre-computing a clique cover $\mathcal{C}$ of G and using the model

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

We refer to this as the *tight* MISP formulation. The clique cover $\mathcal{C}$ is computed using a greedy procedure as follows. Starting with $\mathcal{C} = \emptyset$, 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 $\mathcal{C}$, 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 $n \in \{250, 500, \ldots, 1750\}$ 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 1800 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 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.

We also tested on the 87 instances of the maximum clique problem in the well-known DIMACS benchmark set (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 4 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 relative gap for BDDs and the tight IP model appears in Fig. 6.3(b), 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

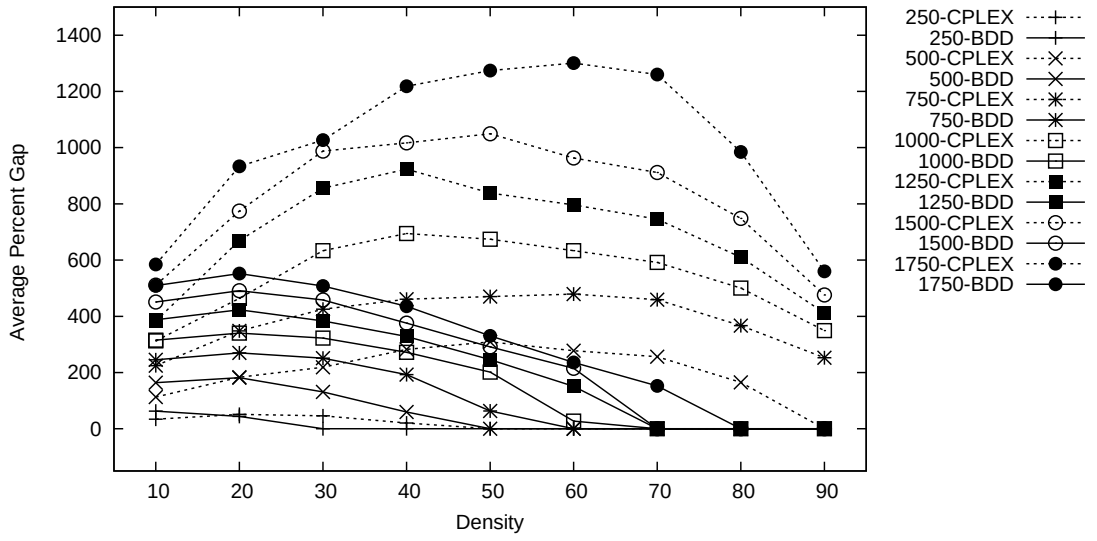
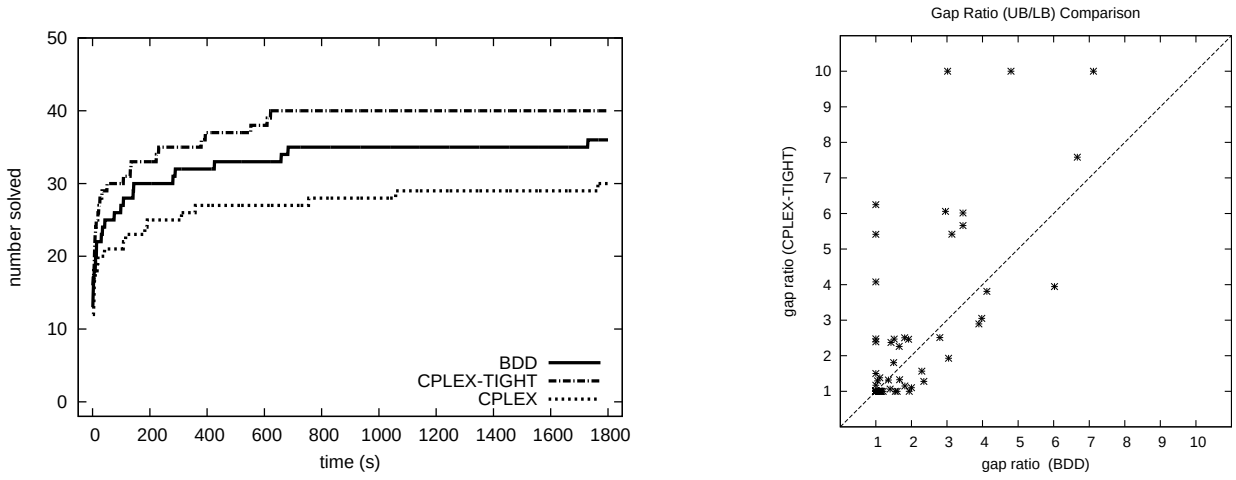


Figure 6.2: Average percent gap on randomly generated MISP instances.



(a) (b)

Figure 6.3: Results on 87 MISP instances for BDDs and CPLEX. (a) Number of instances solved versus time for the tight IP model (top line), BDDs (middle), standard IP model (bottom). (b) End gap comparison after 1800 seconds.

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.

Results for the MCP

We generated random instances of the MCP as follows. 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

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

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

$$\min \left\{ \sum_{(i,j)\in E} w_{ij} y_{ij} \;\middle|\; \begin{Bmatrix} y_{ij} + y_{ik} + y_{jk} \leq 2 \\ y_{ij} + y_{ik} \geq y_{jk} \end{Bmatrix} \text{ all } i, j, k \in \{1, \ldots, n\};\ y_{ij} \in \{0, 1\}, \text{all } (i, j) \in E \right\}$$

We first consider instances with $n = 30$ vertices, all of which were solved by both BDDs and IP within 30 minutes. Figure 6.4 shows 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 Figure 6.5. 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.

Figure 6.6(a) 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 6.6(b) shows that the average percent gap (i.e., $100(\text{UB} - \text{LB})/\text{LB}$) 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.

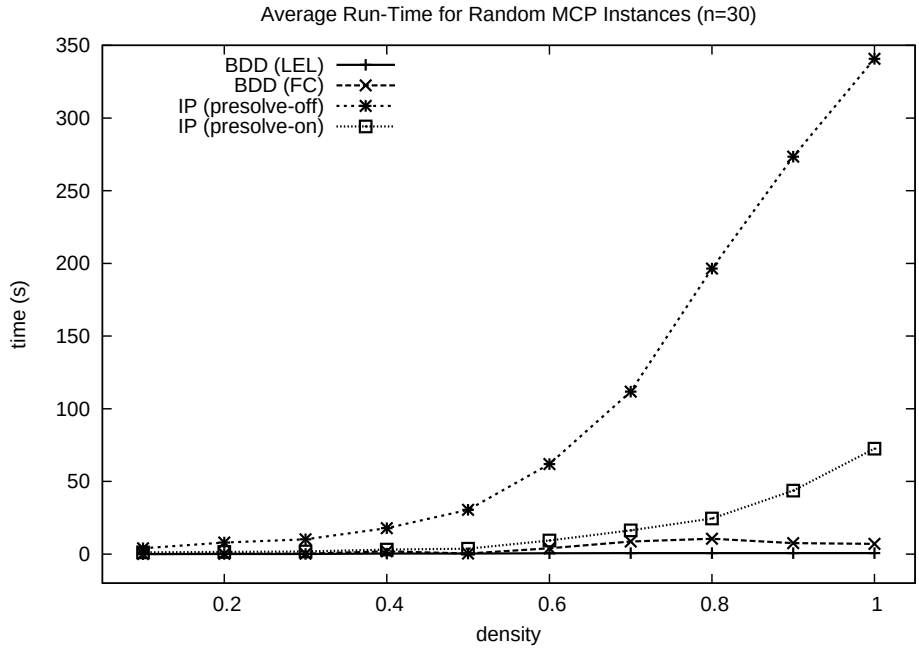


Figure 6.4: Average solution time for MCP instances ($n = 30$ vertices) using BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve). Each point is the average of 10 random instances.

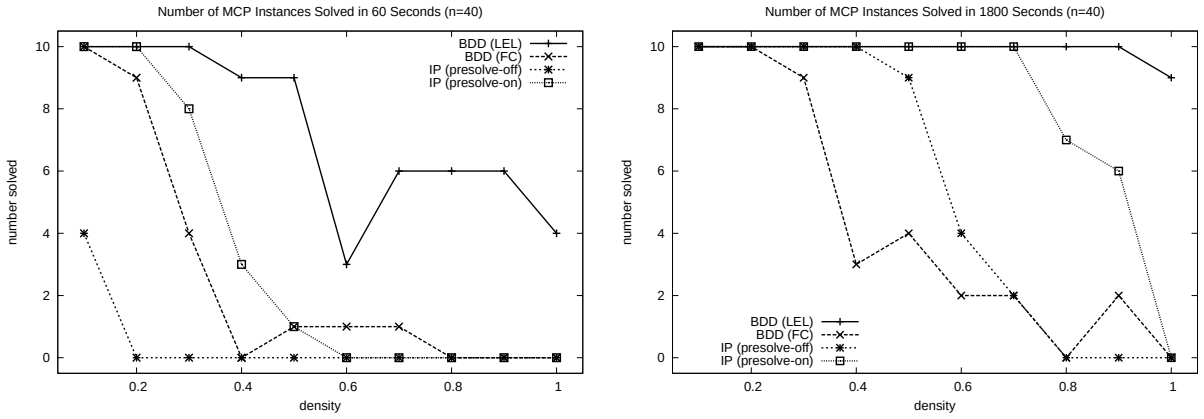
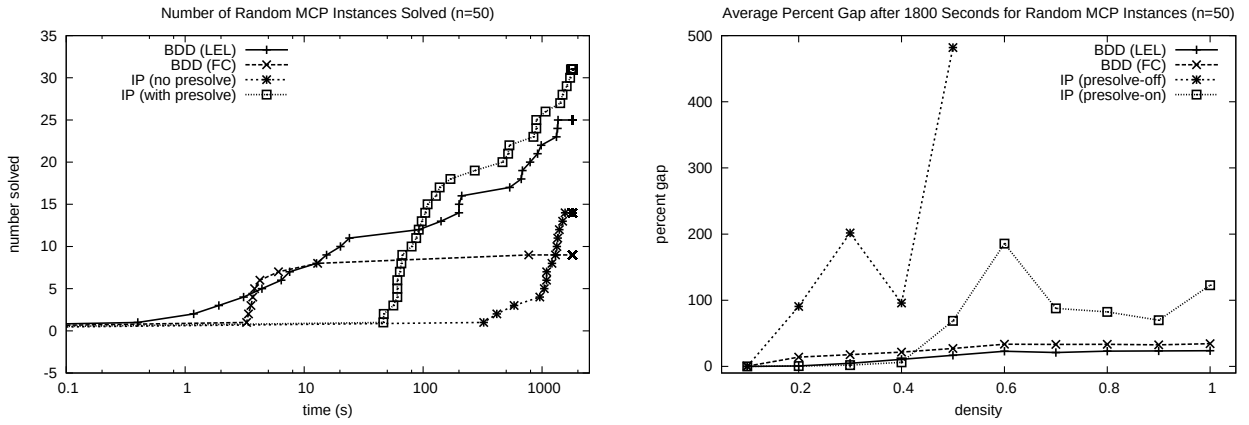


Figure 6.5: Number of MCP instances with $n = 40$ vertices solved after 60 seconds (left) and 1800 seconds (right), versus graph density, using BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve).



(a) (b)

Figure 6.6: (a) Time profile for 100 MCP instances with $n = 50$ vertices, comparing BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve). (b) Percent gap versus density after 1800 seconds, where each point is the average over 10 random instances.

We also tested the algorithm on the g-set, a classical benchmark set, created by the authors in [82], which has since been used extensively for computational testing on algorithms designed to solve the MCP. The 54 instances in the benchmark set are 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 (BK) 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, where the decrease is due to the improvements 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 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	BDD(UB)	BDD(LB)	BK(UB)	BK(LB)	BK(%gap)	NewBK(%gap)	%ReductionInGap
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**

Results for MAX-2SAT

For the MAX-2SAT problem, we created random instances with $n \in \{30, 40\}$ variables and density $d \in \{0.1, 0.2, \ldots, 1\}$. 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_j^i be x_j if x_j is posited in clause i, and $1 - x_j$ if x_j 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

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

Figures 6.7 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

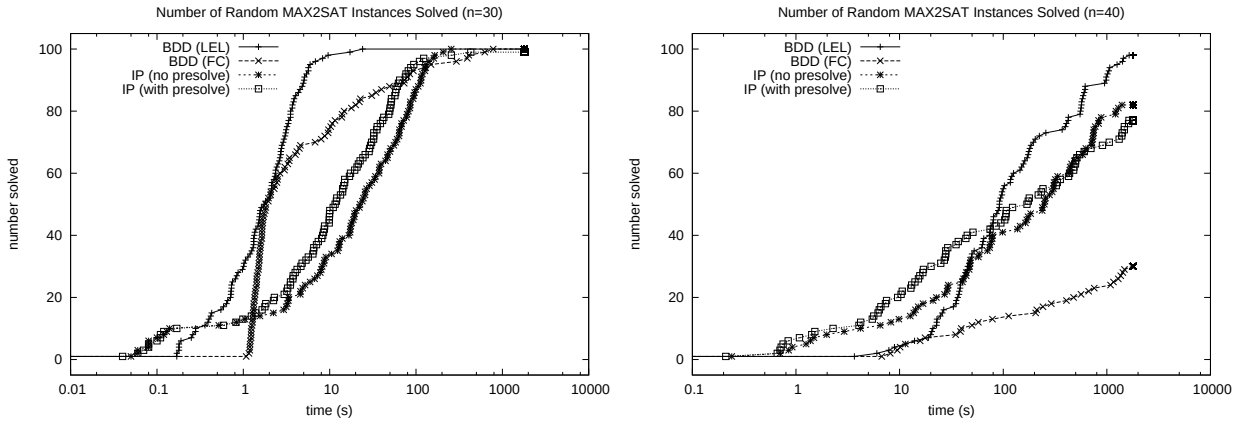


Figure 6.7: Time profile for 100 MAX-2SAT instances with $n = 30$ variables (left) and $n = 40$ variables (right), comparing BDDs (with LEL and FC cutsets) and CPLEX (with and without presolve).

unsolved using no presolve, and 22 unsolved using presolve.

6.3 Parallel Branch-and-bound

In recent years, hardware design has increasingly focused on multi-core 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 sub-trees 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 to using only 1 machine [71]. Furthermore, during the 2011 SAT Competition, the best parallel SAT solvers obtained a average speedup factor of about 3 on 32 cores, which was achieved by employing an algorithm portfolio rather than a parallelized search [92]. In our experimentation, 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

[109], work stealing [40], problem decomposition [116], and a dedicated parallel scheme based on limited discrepancy search [106] all exhibit good speedups (sometimes near-linear) of the CP search in certain settings, especially those involving infeasible instances or scenarios where evaluating search tree leaves is costlier than evaluating internal nodes. Yet, recent developments in CP have moved towards more constraint learning during search, for which efficient parallelization becomes increasingly more difficult.

In general, search schemes relying heavily on learning during search (such as learning new bounds, activities for search heuristics, cuts for MIP, nogoods for CP, and clauses for SAT) tend to be more difficult to efficiently parallelize. *It remains a challenge to design a robust parallelization scheme for solving combinatorial optimization problems* which must necessarily deal with bounds.

Recently, a branch-and-bound scheme based on *approximate decision diagrams* was introduced as a promising alternative to conventional methods (such as integer programming) for solving combinatorial optimization problems [20]. In this paper, our goal is to study how this branch-and-bound search scheme 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 *i)* balancing the workload across the workers, and *ii)* 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 [36, 120, 133], which is a modern programming language designed specifically for building applications for multi-core and clustered systems. For example, [29] 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, IBM ILOG CPLEX 12.5.1 . 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 be each 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.

6.3.1 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 defined as follows. 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.

6.3.2 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 be already 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 matters* — 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.

6.3.3 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 are 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 to those with a larger upper bound) in the exact cut is 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.

6.3.4 Load Balancing

The global queue starts off with a single node corresponding to the root state V, which is 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 $[\ell, u]$.

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

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 $[d, 1]$.

6.3.5 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 [29], 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 6.2.3 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 for, (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

[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.

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.

6.3.6 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 6.2.3. 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 6.2.3 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 non-deterministic ('opportunistic') mode.

DDX10 is implemented using X10 2.3.1 [133] and compiled using the C++ backend 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* [20]. 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 generated random MISP instances as also used previously by **TODO**. We report comparison on instances with 170 vertices

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

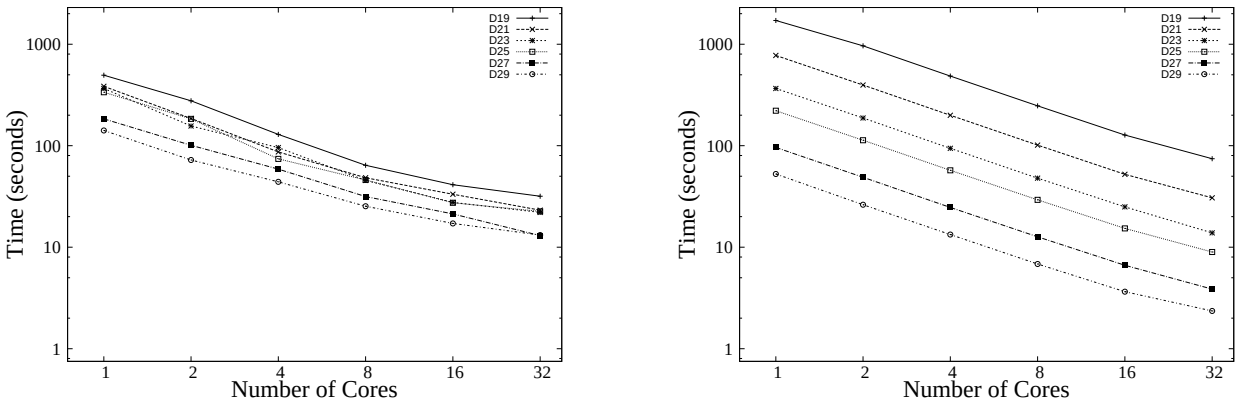


Figure 6.8: Performance of CPLEX (left) and DDX10 (right), with one curve for each graph density ρ shown in the legend as a percentage. Both runtime (y-axis) and number of cores (x-axis) are in log-scale.

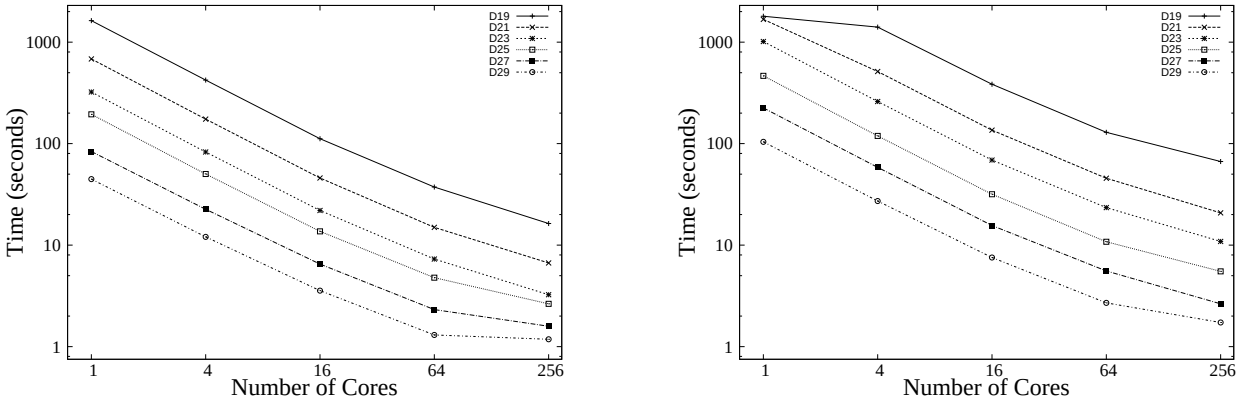


Figure 6.9: Scaling behavior of DDX10 on MISP instances with 170 (left) and 190 (right) vertices, with one curve for each graph density ρ shown in the legend as a percentage. Both runtime (y-axis) and number of cores (x-axis) are in log-scale.

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 6.8 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 left-most 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 by **TODO** for the sequential case, BDD-based branch-and-bound performs better on higher density graphs than sparse graphs. Nevertheless, 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*

Table 6.2: Runtime (seconds) of DDX10 on DIMACS instances. Timeout = 1800.

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	1100.91	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	–	–	1158.18	349.75	172.34
san400_0.9_1.clq	400	0.10	–	–	1386.42	345.66	125.27
san200_0.9_3.clq	200	0.10	–	–	–	487.11	170.08
gen200_p0.9_44.clq	200	0.10	–	–	–	1713.76	682.28
sanr400_0.7.clq	400	0.30	–	–	–	–	1366.98
p_hat700-2.clq	700	0.50	–	–	–	–	1405.46

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.

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 [96], 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.

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.

	1 core		4 cores		16 cores		64 cores		256 cores	
instance	#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	1586	380	1587	392	1511	402	962	224	1028	13
brock200_1.clq	1378	384	1389	393	1396	403	1321	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	1066	2	1068	0	1104	38	1052	13	959	19
san400_0.7_3.clq	–	–	2975	913	2969	916	2789	779	1761	42
p_hat500-2.clq	–	–	2896	710	3011	861	3635	1442	2243	342
p_hat300-3.clq	–	–	–	–	18032	4190	17638	3867	15852	2881
san400_0.9_1.clq	–	–	–	–	2288	238	2218	207	2338	422
san200_0.9_3.clq	–	–	–	–	–	–	9796	390	10302	872
gen200_p0.9_44.clq	–	–	–	–	–	–	43898	5148	45761	7446
sanr400_0.7.clq	–	–	–	–	–	–	–	–	135029	247
p_hat700-2.clq	–	–	–	–	–	–	–	–	89845	8054

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.

As Figure 6.9 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.

DIMACS Instances.

The second experiment is on the DIMACS instances used by [20], where it was demonstrated that sequential BDD-based branch-and-bound has complementary strengths compared to 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 to 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 bounds 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

Application: Sequencing Problems

7.1 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 [62, Section A5]. Sequencing problems are prevalent in manufacturing and routing applications, including production plants where jobs should be processed one at a time in 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 a machine is the bottleneck of a manufacturing plant [111]. 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. Given the practical importance and computational hardness, understanding how sequencing problems can be solved more effectively is an active research area.

In this work we propose 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. In particular, a relaxed MDD can be embedded within a complete search procedure such as branch-and-bound for integer programming or backtracking search for constraint programming [5, 85].

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 to 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 main contributions in this chapter are as follows. We propose a novel formulation of the feasible set of a sequencing problem as an MDD, and show how it can be relaxed so that its size is limited according to a given parameter. We present how the MDD can be used to compute bounds on typical objective functions in scheduling, such as the makespan and total tardiness. Moreover, we demonstrate 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 [10] 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* [14]. 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 [60], Brazilian oil-pipeline scheduling [100], and home health care scheduling [117]. 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. In particular, we were able to close three open TSPLIB instances for the sequencing ordering problem.

The chapter is organized as follows. Section 7.2 presents a brief overview of related literature. Section 7.3 defines the general sequencing problem that will be considered throughout the chapter, and Section 7.4 shows how its feasible set is represented with an MDD. Section 7.5 describes relaxed MDDs and the basic operations for strengthening its representation. Sections 7.6 and 7.7 present detailed methods to filter and refine relaxed MDDs. In Section 7.8, we present an efficient procedure to deduce precedence relations from the relaxation. Section 7.9 demonstrates how the techniques can be used to obtain a polynomial-size MDD that exactly represents the feasible set of a particular TSP variant. Finally, Section 7.10 presents the application of MDDs to constraint-based scheduling, and concluding remarks are given in Section 7.11.

7.2 Related Work

The application of relaxed MDDs to disjunctive scheduling was first proposed by [85], and studied in the context of constraint-based propagators by [43]. Our work expands on the ideas presented in these previous papers, showing new theoretical properties and improved techniques that are applicable to arbitrary sequencing problems.

The techniques we develop here are based on associating a state information with the nodes of the MDD, as in [85]. By doing so, our method is closely related to that of *state-space relaxations* by [39] for routing problems. A similar idea was exploited by [83], which considers a branch-

and-bound algorithm based on homomorphic abstractions of the search space for the sequential ordering problem. In our case, the state-space relaxation is implicitly represented by the nodes of the MDD, which allow us to accommodate multiple constraints more easily in the relaxation. Moreover, we are able to work with different state relaxations simultaneously; namely, one from a top-down perspective of the diagram, and another from a bottom-up perspective, as will become clear in later sections.

Lastly, decision diagrams have also been considered in other areas of optimization, e.g., cut generation in integer programming [15] and 0-1 vertex and facet enumeration [17].

7.3 Problem Definition

In this work 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 $\mathcal{J} = \{j_1, \ldots, j_n\}$ be a set of n jobs to be processed on a machine that can perform at most one job at a time. Each job $j \in \mathcal{J}$ 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 $j, j' \in \mathcal{J}$ 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 $j \in \mathcal{J}$ 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 work: *precedence constraints*, requiring that $s_j \leq s_{j'}$ for certain pairs of jobs $(j, j') \in \mathcal{J} \times \mathcal{J}$, which we equivalently write $j \ll j'$; and *time window constraints*, where the *completion time* $c_j = s_j + p_j$ of each job $j \in \mathcal{J}$ 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 $\max_{j \in \mathcal{J}} c_j$; the *total tardiness*, where we minimize $\sum_{j \in \mathcal{J}} (\max\{0,\ c_j - \delta_j\})$ 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 $\mathcal{J}$. The start time of the 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 $\mathcal{J}$ is *feasible* if the implied job times observe the precedence and time window constraints, and *optimal* if it is feasible and minimizes f.

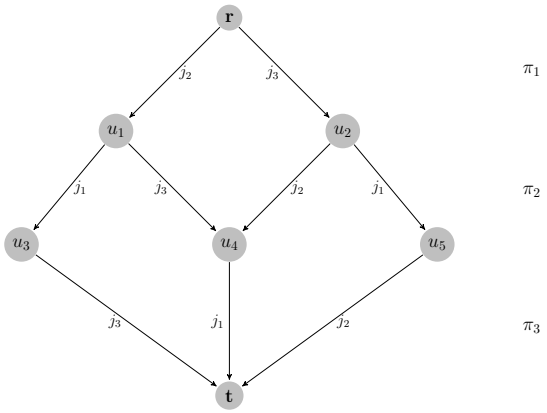
Job Parameters

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.

Figure 7.1: Example of an MDD for a scheduling problem.

7.4 MDD Representation

For the purposes of this chapter, an MDD $\mathcal{M}$ is a directed acyclic graph whose paths represent the feasible orderings of $\mathcal{J}$. The set of nodes of $\mathcal{M}$ are partitioned into $n+1$ layers $L_1, \ldots, L_{n+1}$, where layer L_i corresponds to the i-th position π_i of the feasible orderings encoded by $\mathcal{M}$, for $i = 1, \ldots, n$. Layers L_1 and L_{n+1} are singletons representing the root $\mathbf{r}$ and the terminal $\mathbf{t}$, respectively. 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}, $i \in \{1, \ldots, n\}$. 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 $(a_1, \ldots, a_n)$ from $\mathbf{r}$ to $\mathbf{t}$ identifies the ordering $\pi = (\pi_1, \ldots, \pi_n)$, where $\pi_i = d(a_i)$ for $i = 1, \ldots, n$. 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 20 We provide an MDD representation for a sequencing problem with three jobs j_1, j_2, and j_3. The instance data is presented in Figure 7.1a, and the associated MDD $\mathcal{M}$ is depicted in Figure 7.1b. No precedence constraints are considered. There are 4 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 $\mathbf{r}$. □

Some additional notation follows. The incoming arcs at a node u are denoted by $in(u)$, and the outgoing arcs leaving u by $out(u)$. The *width* of a layer L_i is $|L_i|$, and the width of $\mathcal{M}$ is the maximum width among all layers. The MDD in Figure 7.1b has a width of 3.

Two nodes u, v on the same layer in an MDD are *equivalent* (or belong to the same *equivalence class*) if the set of u-$\mathbf{t}$ paths is equal to the set of v-$\mathbf{t}$ paths. That is, for any u-$\mathbf{t}$ arc-specified path $(a_1, a_2, \ldots, a_k)$ there exists a v-$\mathbf{t}$ arc-specified path $(a'_1, a'_2, \ldots, a'_k)$ such that $d(a_1) = d(a'_1)$, $d(a_2) = d(a'_2)$, $\ldots$, $d(a_k) = d(a'_k)$, and vice-versa. An MDD $\mathcal{M}$ is *reduced* if no two nodes in any layer are equivalent. This is the case, for example, for the MDD in Figure 7.1b. A standard result in decision diagram theory is that there exists a unique reduced MDD representing the feasible orderings of $\mathcal{J}$, provided that we do not change the mapping between the layers L_i of $\mathcal{M}$ and the ordering positions π_i; see, e.g, [132]. The reduced MDD also has the smallest width among the MDDs encoding the feasible orderings of $\mathcal{J}$.

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 $\mathcal{M}$. 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 $\mathcal{M}$.

- *Makespan.* For each arc a in $\mathcal{M}$, 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 $\mathcal{M}$ 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

$$ect_a = \max\{r_{d(a)},\ \min\{ect_{a'} + t_{d(a'),d(a)} : a' \in in(u)\}\} + p_{d(a)}. \tag{7.1}$$

  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 $\mathcal{M}$. An optimal ordering can be obtained by recursively retrieving the minimizer arc $a' \in in(u)$ in the "min" of (7.1).

- *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 $\mathcal{M}$ containing a. If a is directed out of $\mathbf{r}$, we have $st_a = 0$; otherwise,

$$st_a = \min\{st_{a'} + t_{d(a'),d(a)} : a' \in in(u)\}. \tag{7.2}$$

  The minimum sum of of setup times is given by $\min_{a \in in(\mathbf{t})} st_a$.

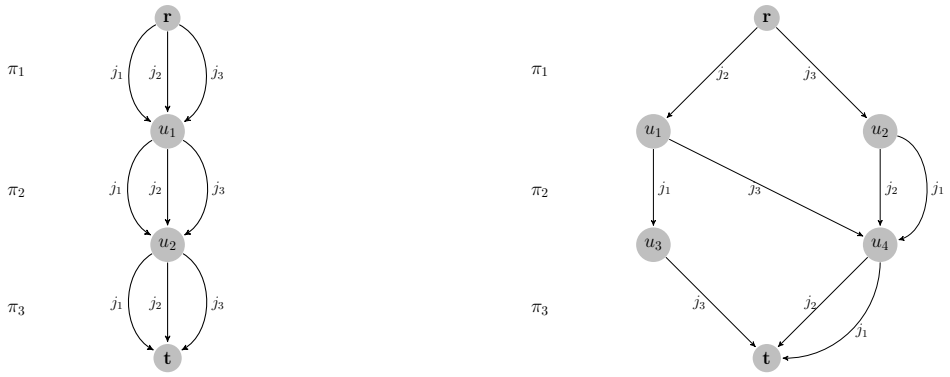
- *Total Tardiness.* The *tardiness* of a job j is defined by $\max\{0,\ c_j - \delta_j\}$ for some due date δ_j. Unlike the previous two cases, the tardiness value that a job attains in an optimal solution depends on the sequence of all activities, not only on its individual contribution or the value of its immediate predecessor. Nonetheless, as the tardiness function for a job is non-decreasing 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 $\mathcal{M}$ 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 $\sum_{j\in\mathcal{J}} f_j(c_j)$, where f_j is a function defined for each job j and which is non-decreasing on 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 $\mathcal{M}$ containing a. Using such bounds as arc lengths, the shortest path from $\mathbf{r}$ to $\mathbf{t}$ represents a lower bound on $\sum_{j\in\mathcal{J}} f_j(c_j)$. This bound is tight if $f_j(c_j) = c_j$, or if $\mathcal{M}$ 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 21 In the instance depicted in Figure 7.1, we can apply the recurrence relation (7.1) 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 on the tardiness of j_3. □



(a) 1-width relaxation.

(b) 2-width relaxation.

Figure 7.2: Two relaxed MDDs for the sequencing problem in Figure 7.1.

7.5 Relaxed MDDs

A *relaxed MDD* is an MDD $\mathcal{M}$ that represents a superset of the feasible orderings of $\mathcal{J}$; i.e., every feasible ordering is identified by some path in $\mathcal{M}$, but not necessarily all paths in $\mathcal{M}$ identify a feasible ordering. We construct relaxed MDDs by limiting the size to a fixed maximum allowed width W. Thus, the strength of the relaxed MDD can be controlled by increasing W; we obtain an exact MDD by setting W to infinity.

Figures 7.2a and 7.2b present two examples of a relaxed MDD with maximum width $W = 1$ and $W = 2$, respectively, for the problem depicted in Figure 7.1. In particular, the MDD in Figure 7.2a encodes all the orderings represented by permutations of $\mathcal{J}$ 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}, $i = 1, \ldots, n$, with arcs $a_1, \ldots, a_n$ such that $d(a_l) = j_k$ for each job j_k.

It can also be verified that the MDD in Figure 7.2b contains all the feasible orderings of the instance in Figure 7.1. However, the right-most path going trough nodes $\mathbf{r}$, u_2, u_4, and $\mathbf{t}$ 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 7.4 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 $\mathcal{J}$ are encoded in the diagram. Moreover, the lower bounding technique for total tardiness remains valid.

Considering that a relaxed MDD $\mathcal{M}$ can be easily constructed for any sequencing problem (e.g., the 1-width relaxation of Figure 7.2a), we will now recall the techniques presented in Section 4.3 to modify $\mathcal{M}$ in order to strengthen the relaxation it provides while observing the maximum width W. These are based on the compilation procedures developed by [78] and [85] for general constraint satisfaction systems. Under certain conditions, we obtain the reduced MDD representing exactly the feasible orderings of $\mathcal{J}$, provided that W is sufficiently large.

Namely, we modify a relaxed MDD $\mathcal{M}$ by applying the operations of *filtering* and *refinement*, which aim at approximating $\mathcal{M}$ to an *exact* MDD, i.e., one that exactly represents the feasible orderings of $\mathcal{J}$. They are described as follows.

- *Filtering.* We write that an arc a is *infeasible* if all the paths in $\mathcal{M}$ containing a represent orderings that are not feasible. Filtering consists of identifying infeasible arcs and removing them from $\mathcal{M}$, which would hence eliminate one or more infeasible orderings that are encoded in $\mathcal{M}$. We will provide details on the filtering operation in Section 7.6.

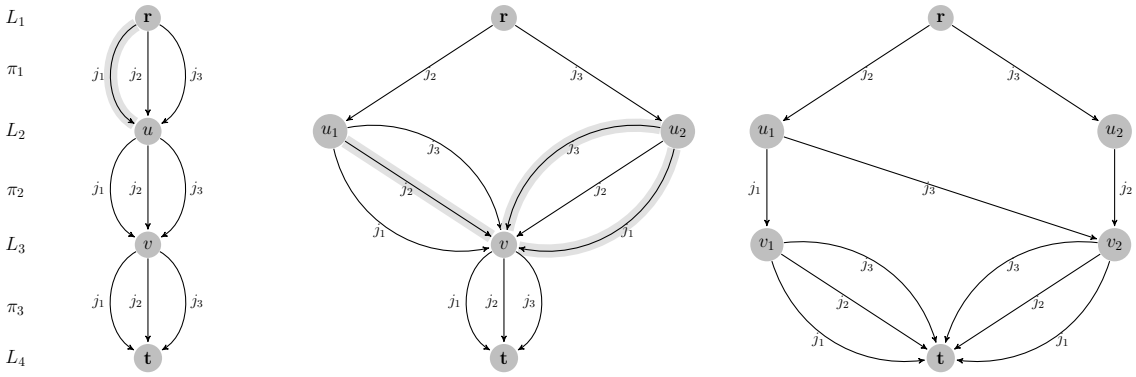
- *Refinement.* A relaxed MDD can be intuitively perceived as a diagram obtained by merging non-equivalent nodes of an exact MDD for the problem. Refinement consists of identifying these nodes in $\mathcal{M}$ that are encompassing multiple equivalence classes, and *splitting* them into two or more new 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 $\mathcal{M}$ representing such orderings must end in different nodes of the MDD, which will be necessarily non-equivalent by definition. We will provide details on the refinement operation in Section 7.7.

Observe that if a relaxed MDD $\mathcal{M}$ does not have any infeasible arcs and no nodes require splitting, then by definition $\mathcal{M}$ is exact. However, it may not necessarily be reduced.

Filtering and refinement are independent operations that can be applied to $\mathcal{M}$ in any order that is suitable for the problem at hand. In this work 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 22 Figure 7.3 illustrates the top-down application of filtering and refinement for layers L_2 and L_3. Assume a scheduling problem with three jobs $\mathcal{J} = \{j_1, j_2, j_3\}$ and subject to a single precedence constraint stating that job j_2 must precede job j_1. The initial relaxed MDD is an 1-width relaxation depicted in Figure 7.3a. 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 Figure 7.3b. Note that when a node is split, we replicate its outgoing arcs to each of the new nodes.



(a) Initial relaxation. (b) After processing L_2. (c) After processing L_3.

Figure 7.3: 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.

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 $\mathbf{r}$ 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 Figure 7.3c. □

7.6 Filtering

In this section we apply a methodology derived from [5] and [85] to identify necessary conditions for the infeasibility of an arc in $\mathcal{M}$. This is done as follows. For each constraint type $\mathcal{C}$, we equip the nodes and arcs of $\mathcal{M}$ with a *state information* $\mathtt{s}_\mathcal{C}$. Each state $\mathtt{s}_\mathcal{C}$ is then considered separately to identify conditions that deem an arc as infeasible according to the particular structure of $\mathcal{C}$. Note that, in general, we are not able to derive efficient infeasibility conditions that are necessary and sufficient for all constraints of the problem (if $P \neq NP$), since it is NP-hard to decide if there is a feasible solution to a scheduling problem with arbitrary release dates and deadlines.

The tests presented here can be computed in polynomial-time in the size of the relaxed MDD $\mathcal{M}$. Namely, we restrict our state definitions to those with size $O(|\mathcal{J}|)$ and that satisfy a *Markovian* property, 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 7.5. We also describe additional states that are obtained through an extra bottom-up traversal of the MDD and that, when combined with the top-down states, lead to stronger tests.

7.6.1 Filtering invalid permutations

The feasible orderings of any sequencing problem are permutations of $\mathcal{J}$ without repetition, which can be perceived as an implicit constraint to be observed. Here we can directly use the filtering conditions described in Section 4.3.1. Namely, let us associate two states $All_u^{\downarrow} \subseteq \mathcal{J}$ and $Some_u^{\downarrow} \subseteq \mathcal{J}$ to each node u of $\mathcal{M}$. 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. For example, in Figure 7.3b without the shaded arcs, $All_v^{\downarrow} = \{j_2\}$ and $Some_v^{\downarrow} = \{j_1, j_2, j_3\}$ for node v.

We trivially have $All_{\mathbf{r}}^{\downarrow} = Some_{\mathbf{r}}^{\downarrow} = \emptyset$. Furthermore, 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

$$All_v^{\downarrow} = \bigcap_{a=(u,v)\in in(v)} (All_u^{\downarrow} \cup \{d(a)\}), \tag{7.3}$$

$$Some_v^{\downarrow} = \bigcup_{a=(u,v)\in in(v)} (Some_u^{\downarrow} \cup \{d(a)\}). \tag{7.4}$$

As presented in Section 4.3.1, an arc $a = (u, v)$ is infeasible if either $d(a) \in All_u^{\downarrow}$ (condition 4.4) or $|Some_u^{\downarrow}| = \ell(a)$ and $d(a) \in Some_u^{\downarrow}$ (condition 4.5).

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 $All_u^{\uparrow} \subseteq \mathcal{J}$ and $Some_u^{\uparrow} \subseteq \mathcal{J}$ for each node u of $\mathcal{M}$. They are equivalent to the states $All_u^{\downarrow}$ and $Some_u^{\downarrow}$, but now they are computed with respect to the paths from $\mathbf{t}$ to u instead of the paths from $\mathbf{r}$ to u. As before, they are recursively obtained through the relations

$$All_u^{\uparrow} = \bigcap_{a=(u,v)\in out(u)} (All_v^{\uparrow} \cup \{d(a)\}), \tag{7.5}$$

$$Some_u^{\uparrow} = \bigcup_{a=(u,v)\in out(u)} (Some_v^{\uparrow} \cup \{d(a)\}), \tag{7.6}$$

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

It follows from Section 4.3.1 that an arc $a = (u, v)$ is infeasible if either $d(a) \in All_v^{\uparrow}$ (condition 4.8), $|Some_v^{\uparrow}| = n - \ell(a)$ and $d(a) \in Some_v^{\uparrow}$ (condition 4.9), or $|Some_u^{\downarrow} \cup \{d(a)\} \cup Some_v^{\uparrow}| < n$ (condition 4.10).

7.6.2 Filtering precedence constraints

Suppose now we are given a 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 in Section 7.6.1 for this particular case.

Lemma 24 *An arc $a = (u, v)$ is infeasible if any of the following conditions hold:*

$$\exists\, j \in (\mathcal{J} \setminus Some_u^{\downarrow}) \;\; s.t. \;\; j \ll d(a), \tag{7.7}$$

$$\exists\, j \in (\mathcal{J} \setminus Some_v^{\uparrow}) \;\; s.t. \;\; d(a) \ll j. \tag{7.8}$$

Proof. Let π' be any partial ordering identified by a path from $\mathbf{r}$ to u, and consider (7.7). By definition of $Some_u^{\downarrow}$, we have that any job j in the set $(\mathcal{J} \setminus Some_u^{\downarrow})$ is not assigned to any position in π'. Thus, if any of such jobs 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 (7.8) is the symmetrical version of (7.7). ■

7.6.3 Filtering time window constraints

Consider now that a deadline d_j is imposed for each job $j \in \mathcal{J}$. With each arc a we associate the state ect_a as defined in Section 7.4: 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 $\mathcal{M}$ containing the arc a. As in relation (7.1), the state ect_a for an arc $a = (u, v)$ is given by the recurrence

$$ect_a = \begin{cases} r_{d(a)} + p_{d(a)} & \text{if } a \in out(\mathbf{r}), \\ \max\{r_{d(a)},\ \min\{ect_{a'} + t_{d(a'),d(a)} : a' \in in(u), d(a) \neq d(a')\}\} + p_{d(a)} & \text{otherwise,} \end{cases}$$

where we added the trivial condition $d(a) \neq d(a')$ to strengthen the bound on the time above. 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 $\mathcal{M}$ 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 $\mathcal{M}$:

$$lst_a = \begin{cases} d_{d(a)} - p_{d(a)} & \text{if } a \in in(\mathbf{t}), \\ \min\{d_{d(a)},\ \max\{lst_{a'} - t_{d(a),d(a')} : a' \in out(v), d(a) \neq d(a')\}\} - p_{d(a)} & \text{otherwise.} \end{cases}$$

Lemma 25 *An arc $a = (u, v)$ is infeasible if*

$$ect_a > lst_a + p_{d(a)}. \tag{7.9}$$

Proof. The value $lst_a + p_{d(a)}$ represents an upper bound on the the maximum time the job $d(a)$ can be completed so that no deadlines are violated in the orderings identified by paths in $\mathcal{M}$ 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 (7.9) holds. ■

7.6.4 Filtering objective function bounds

Let z^* be an upper bound of the objective function value (e.g., corresponding to the best feasible solution found during the search for an optimal solution). Given z^*, an arc a is infeasible with respect to the objective if all paths in $\mathcal{M}$ that contain a have objective value greater than z^*. However, the associated filtering method depends on the form of the objective function. For example, 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 consider the same infeasibility condition in Lemma 25.

If z^* corresponds to an upper bound on the sum of setup times, we proceed as follows. For each arc $a = (u, v)$ in $\mathcal{M}$, 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 have

$$st_a^{\downarrow} = \begin{cases} 0, & \text{if } a \in out(\mathbf{r}), \\ \min\{t_{d(a'),d(a)} + st_{a'}^{\downarrow} : a' \in in(u), d(a) \neq d(a')\}, & \text{otherwise.} \end{cases}$$

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 $st_a^{\uparrow}$ is given below, computed through a bottom-up traversal of $\mathcal{M}$:

$$st_a^{\uparrow} = \begin{cases} 0, & \text{if } a \in in(\mathbf{t}), \\ \min\{t_{d(a),d(a')} + st_{a'}^{\uparrow} : a' \in out(v), d(a) \neq d(a')\}, & \text{otherwise.} \end{cases}$$

Lemma 26 *An arc a is infeasible if*

$$st_a^{\downarrow} + st_a^{\uparrow} > z^*. \tag{7.10}$$

Proof. It follows directly from the definitions of $st_a^{\downarrow}$ and $st_a^{\uparrow}$. ■

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 27 *A node u should be removed from $\mathcal{M}$ if*

$$sp_u^{\downarrow} + sp_u^{\uparrow} > z^*, \tag{7.11}$$

Proof. Length l_a represents a lower bound on the tardiness of job $d(a)$ with respect to solutions identified by **r**-**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 **r** to u and **t** to u, respectively, since the tardiness of a job is non-decreasing on its completion time. ■

7.7 Refinement

Recall from Section 7.5 that a relaxed MDD can be strengthened by a refinement operation. Ideally, refinement should modify a layer so that each of its nodes exactly represents a particular equivalence class. However, as it may be necessary to create an exponential number of nodes to represent all equivalence classes, we apply in this section a heuristic refinement procedure that observes the maximum width W when creating new nodes in a layer.

We consider the refinement heuristic described in Section 4.3.1 for all applications in this chapter. Our goal 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 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 described in Section 7.6.1, 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.

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 a very large release date and processing times in comparison to 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 7.10. Recall that the refinement heuristic requires a *ranking* of jobs $\mathcal{J}^* = \{j_1^*, \ldots, j_n^*\}$, where jobs with smaller index in $\mathcal{J}^*$ have a higher priority.

We note that the refinement heuristic also yields a *reduced* MDD $\mathcal{M}$ for certain structured problems, given a sufficiently large width. The following corollary, stated without proof, is directly

derived from Lemma 20 and Theorem 19.

COROLLARY 28 *Assume $W = +\infty$. For a sequencing problem having only precedence constraints, the relaxed MDD $\mathcal{M}$ that results from the constructive proof of Theorem 19 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 $\mathcal{M}$ is still a valid relaxation and may provide a stronger representation.

7.8 Inferring Precedence Relations from Relaxed MDDs

Given a set of precedence relations to a problem (e.g., that were possibly derived from other relaxations), we can use the filtering rules (7.7) and (7.8) from Section 7.6.2 to strengthen a relaxed MDD. In this section, we show that a converse relation is also possible. Namely, given a relaxed MDD $\mathcal{M}$, we can deduce all precedence relations that are satisfied by the partial orderings represented by $\mathcal{M}$ in polynomial time in the size of $\mathcal{M}$. To this end, assume that the states $All_u^{\downarrow}$, $All_u^{\uparrow}$, $Some_u^{\downarrow}$, and $Some_u^{\uparrow}$ as described in Section 7.6.1 are computed for all nodes u in $\mathcal{M}$. We have the following results.

THEOREM 29 *Let $\mathcal{M}$ be an MDD that exactly identifies all the feasible orderings of $\mathcal{J}$. A job j must precede job j' in any feasible ordering if and only if $(j' \notin All_u^{\downarrow})$ or $(j \notin All_u^{\uparrow})$ for all nodes u in $\mathcal{M}$.*

Proof. Suppose there exists a node u in layer L_i, $i \in \{1, \ldots, n+1\}$, such that $j' \in All_u^{\downarrow}$ and $j \in All_u^{\uparrow}$. By definition, there exists a path $(\mathbf{r}, \ldots, u, \ldots, \mathbf{t})$ that identifies an ordering where job j' starts before job j. This can only be true if and only if job j does not precede j' in any feasible ordering. ■

COROLLARY 30 *The set of all precedence relations that must hold in any feasible ordering can be extracted from $\mathcal{M}$ in $O(n^2\,|\mathcal{M}|)$.*

Proof. Construct a digraph $G^* = (\mathcal{J}, E^*)$ by adding an arc (j, j') to E^* if and only if there exists a node u in $\mathcal{M}$ such that $j' \in All_u^{\downarrow}$ and $j \in All_u^{\uparrow}$. Checking this condition for all pair of jobs takes $O(n^2)$ for each node in $\mathcal{M}$, and hence the time complexity to construct G^* is $O(n^2|\mathcal{M}|)$. According to Theorem 29 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 29.

COROLLARY 31 *Given a relaxed MDD $\mathcal{M}$, 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 $\mathcal{M}$.*

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 29 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 31, the precedence relations implied by the solutions of a relaxed MDD $\mathcal{M}$ can be extracted by applying the algorithm in Corollary 30 to the states $Some_v^{\downarrow}$ and $Some_v^{\uparrow}$. Since $\mathcal{M}$ has at most $O(n\,W)$ nodes and $O(n\,W^2)$ 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.

7.9 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 of such cases for a problem class introduced by [10], 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 $j_p, j_q \in \mathcal{J}$ if $q \geq p+k$. This precedence structure was motivated by a real-world application in steel rolling mill scheduling. The work by [12] also demonstrates how solution methods to this class of problems can serve as auxiliary techniques in other cases, for example as heuristics for the traveling salesman problem and vehicle routing with time windows.

We stated in Corollary 28 that we are able to construct the reduced MDD $\mathcal{M}$ when only precedence constraints are imposed and a sufficiently large W is given. We have the following results for $\mathcal{M}$ if the precedence relations satisfy the discrepancy structure for a given k.

Lemma 32 *For a node $v \in L_{m+1}$, $m = 1, \ldots, n$, we have $All_v^{\downarrow} \subseteq \{j_1, \ldots, j_{\min\{m+k-1,\, n\}}\}$.*

Proof. If $m+k-1 > n$ we obtain the redundant condition $All_u^{\downarrow} \subseteq \mathcal{J}$, 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 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 33 *The width of $\mathcal{M}$ 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 $\mathcal{F} := \{All_u^{\downarrow} : u \in L_{m+1}\}$. It can be shown that if $\mathcal{M}$ is reduced, no two nodes $u, v \in L_{m+1}$ are such that $All_u^{\downarrow} = All_v^{\downarrow}$. Thus, $|\mathcal{F}| = |L_{m+1}|$.

We derive the cardinality of $\mathcal{F}$ as follows. Take $All_v^{\downarrow} \in \mathcal{F}$ 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 32, 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 $\mathcal{F}$ 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 of such set $All_u^{\downarrow}$ contain the jobs $j_1, \ldots, j_{m-l-1}$ according to Lemma 32. Hence, the remaining $m-(m-l-1)-1 = l$ job indices can be freely chosen from $m-l, \ldots, m+k-l-2$. Notice there are no imposed precedences on these remaining $m+k-l-2-(m-l)+1 = k-1$ elements; thus, there exist $\binom{k-1}{l}$ of such subsets. But these sets define a partition of $\mathcal{F}$. Therefore

$$|\mathcal{F}| = |L_{m+1}| = \sum_{l=0}^{k-1} \binom{k-1}{l} = \binom{k-1}{0} + \cdots + \binom{k-1}{k-1} = 2^{k-1}.$$

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 will have less than $k-1$ possibilities for the new combinations, and hence the maximum number of nodes is strictly less than 2^{k-1} for these cases. The width of $\mathcal{M}$ is therefore 2^{k-1}. ■

According to Theorem 33, $\mathcal{M}$ 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 $2^{k-1}.2^{k-1} = 2^{2k-2}$ arcs directed out of a layer). Using the recursive relation (7.2) in Section 7.4, 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 [10] provides an algorithm that minimizes this same function in $O(n\,k^2\,2^{k-2})$, but that is restricted to this particular objective.

7.10 Application to Constraint-based Scheduling

We added the techniques described here to *ILOG CP Optimizer* (CPO), the current state-of-the-art general-purpose scheduler. Given a sequencing problem as considered in this work, 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 *propagators* 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 [14] and [130].

We have implemented our techniques as a user-defined propagator, which maintains a relaxed MDD and runs one round of top-down filtering and refinement when activated at each node of the branching tree. 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 7.8, which may trigger additional propagation by CPO. Our implementation follows the guidelines from [91].

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.

7.10.1 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 work, 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 2.33GHz with 8 Gb RAM. The MDD code was implemented in C++ using the CPO callable library from the ILOG CPLEX Academic Studio V.12.4.01. We have 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`.

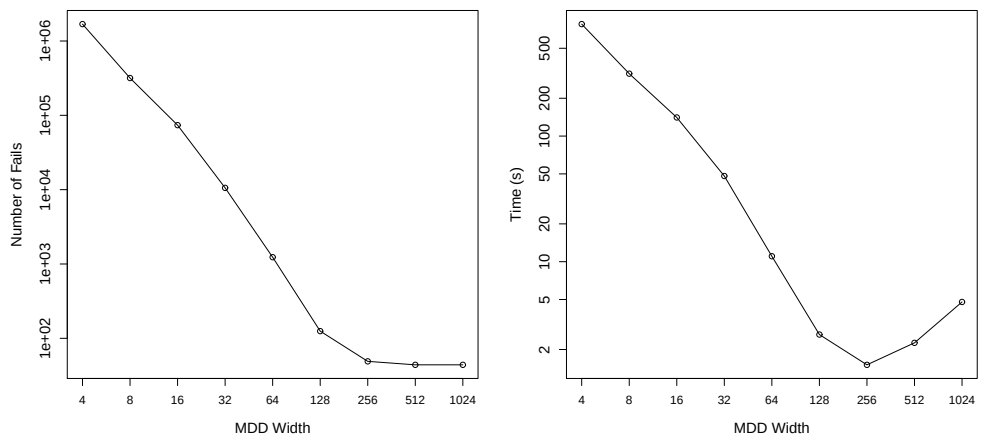
7.10.2 Impact of the MDD Parameters

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 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 [64], 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 7.7: 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. $j_i^* = \arg\max_{p \in \mathcal{J} \setminus \{j_1^*, \ldots, j_{i-1}^*\}} \{\sum_{k=1}^{i-1} t_{j_k^*, p}\}$ for the setup times t. The intuition is that we want jobs with largest travel distances to be exactly represented in $\mathcal{M}$.

The number of fails and total time to find the optimal solution for different MDD widths are presented in Figure 7.4. Due to the properties of the refinement technique in Theorem 19, we consider only powers of 2 as widths. We note from Figure 7.4a that the number of fails is decreasing rapidly as the width increases, up to a point where it becomes close to a 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 Figure 7.4b. Namely, the times decrease accordingly as the width increases. At the point where the relaxed MDD is close to be 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

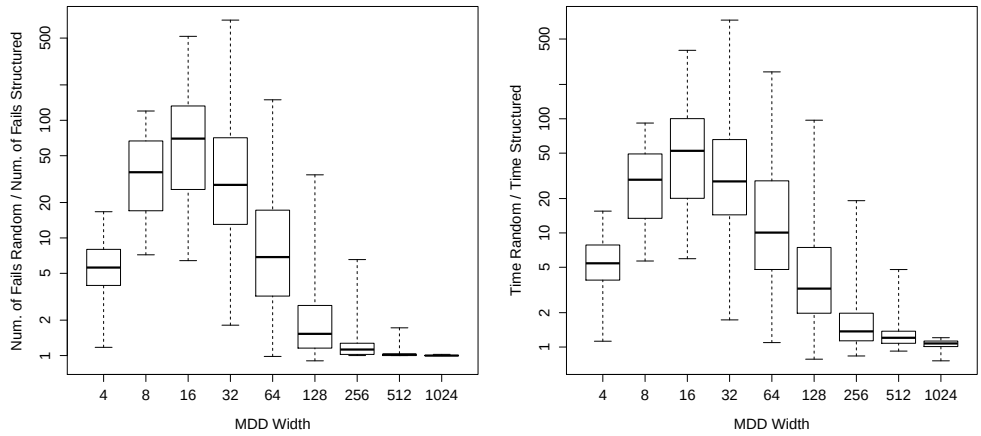


(a) Number of fails. (b) Time.

Figure 7.4: 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.

for the refinement strategy described in Section 7.7. 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 Figure 7.5. 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 Figure 7.5 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 19. Thus, distinct refinement strategies are not expected to differ much on average, as shown, e.g., in the width-4 case of Figure 7.5a. 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 on 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 Figure 7.5a). Another aspect we observe in Figure 7.5b 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.



(a) Number of fails ratio.

(b) Time ratio.

Figure 7.5: Performance comparison between random and structured refinement strategies for the TSPTW instance `n20w200.001`. The axes are in logarithm scale.

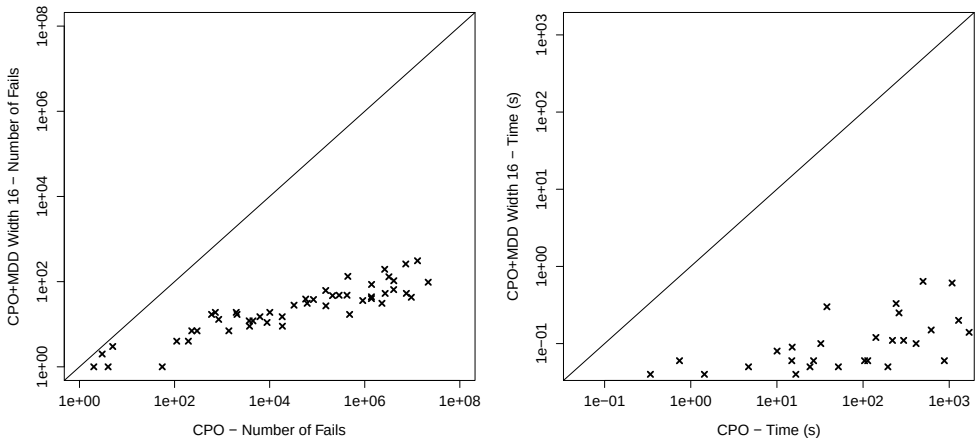
7.10.3 Traveling Salesman Problem with Time Windows

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 [64], [48], and [8], 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 to all methods, and we used the structured job ranking described in Section 7.10.2.

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 Figure 7.6, 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 to 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 7.7a compares the times for instances solved by both methods, while Figure 7.7b depicts the performance plot. In



(a) Number of fails. (b) Time.

Figure 7.6: 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.

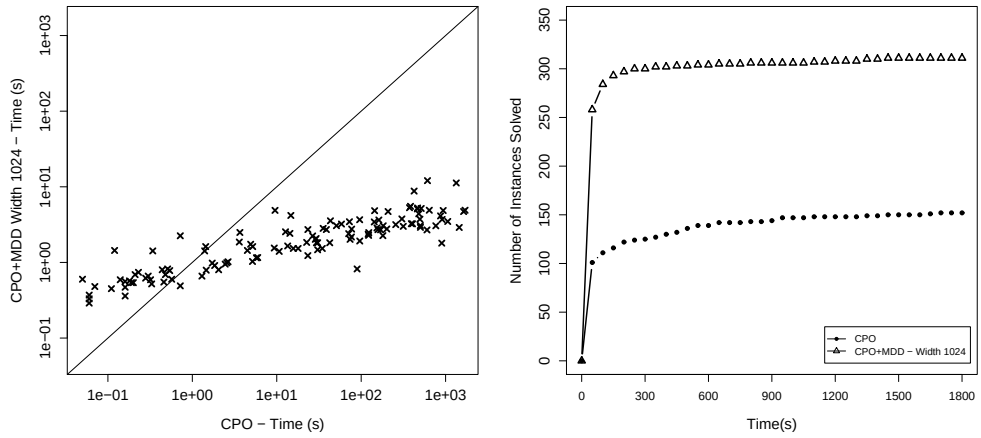
particular, the overhead introduced by the MDD is only considerable for small instances (up to 20 jobs). On the majority of the cases, the `CPO+MDD` is capable of proving optimality much quicker.

7.10.4 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 *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 less than 70 vertices from the well-known [126] benchmark, proposed initially by [9], 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` model 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 7.10.2. The results are reported in Table 7.1. The column *Best* corresponds to the best solution found by the method and the column *Time* corresponds to the time that the solution was proved optimal. A value TL



(a) Scatter plot.

(b) Performance plot.

Figure 7.7: 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.

indicates that the time limit was reached. Since the TSPLIB results are not updated, we report updated bounds obtained from [83], [68], and [6].

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 [83] (for a computer approximately 10 times slower than ours), and by more than 4 hours by [68] (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 7.1. On the other hand, less constrained instances are better suited to MILP-based approaches; instances p43.1 and ry48p.1 are solved by a few second in [9].

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 and the ones provided by [9], from where these problems were proposed. This includes the instance p43.1 which was solved in that work.

7.10.5 Makespan Problems

Constraint-based solvers are known to be particularly effective when the objective function is makespan, which is greatly due to specialized domain propagation techniques that can be used in such cases; see, e.g., [14].

In this section we evaluate the performance of `CPO` and `CPO+MDD` on sequencing problems with

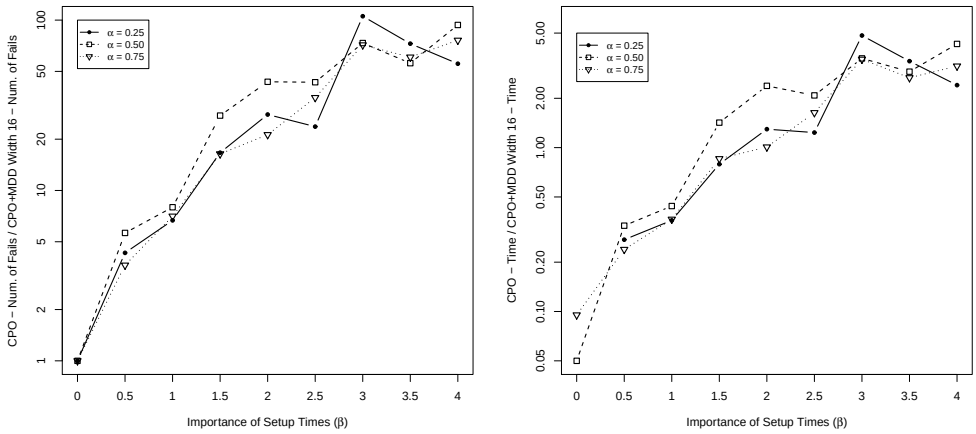
instance	vertices	bounds	CPO best	CPO time (s)	CPO+MDD, width 2048 best	CPO+MDD, width 2048 time (s)
br17.10	17	55	55	0.01	55	4.98
br17.12	17	55	55	0.01	55	4.56
ESC07	7	2125	2125	0.01	2125	0.07
ESC25	25	1681	1681	TL	1681	48.42
p43.1	43	28140	28205	TL	28140	287.57
p43.2	43	[28175, 28480]	28545	TL	**28480**	**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

Table 7.1: Results on ATSPP instances. Values in bold represent instances solved for the first time.

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 [35], 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 $\{0, \ldots, (50.5)\beta\}$, 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 Figure 7.8. For each value of α, we plot the ratio of `CPO` and `CPO+MDD` in terms of the number of fails (Figure 7.8a) and time (Figure 7.8b). The plot in Figure 7.8a indicates that the `CPO+MDD` inference becomes more dominant in comparison to `CPO` for larger values of β, that is, when setup times become more important. The MDD introduces a computational overhead in comparison to 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 to `CPO`. The same behavior was observed on average for other



(a) Number of fails ratio.

(b) Time ratio.

Figure 7.8: Comparison between CPO and CPO+MDD for minimizing makespan on three instances with randomly generated setup times. The vertical axes are in logarithmic scale.

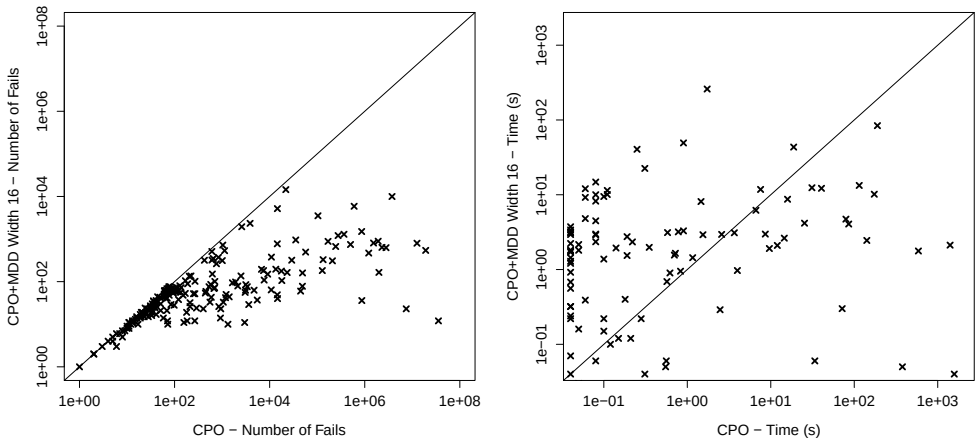
base instances generated under the same scheme.

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 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 was imposed to 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 Figure 7.9, 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 a better performance for harder instances; the lower diagonal of the Figure 7.9b 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 7.10.3.

7.10.6 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



(a) Number of fails. (b) Time.

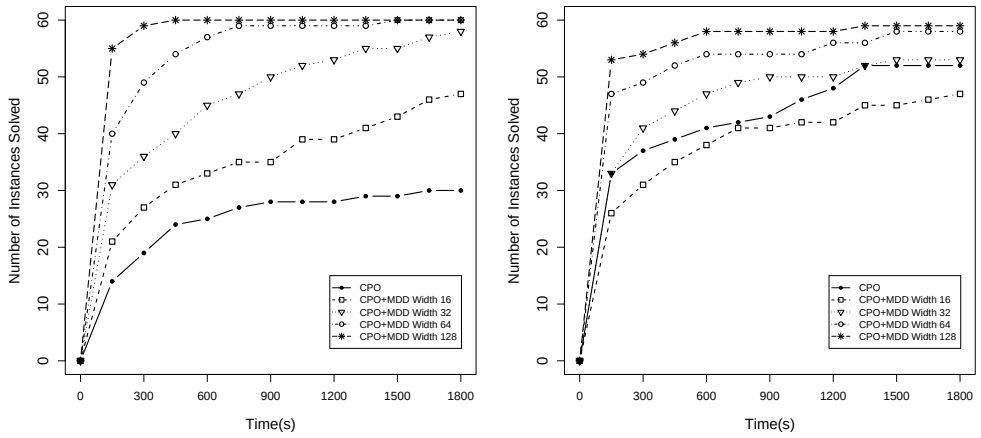
Figure 7.9: Performance comparison between `CPO` and `CPO+MDD` for minimizing makespan on Dumas and Gendreau TSPTW classes. The vertical and horizontal axes are in logarithmic scale.

by [14]. 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 [35]. 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 $\{0, \ldots, \alpha \sum_i p_i\}$, and the due dates are selected uniformly at random from the set $\{r_i + p_i, \ldots, r_i + p_i + \beta \sum_i p_i\}$. 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 $\{1, \ldots, 10\}$. We generated 5 instances for each configuration, hence 120 instances in total. A time limit of 1,800 seconds was imposed to 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 Figure 7.10a, and the results for the weighted total tardiness instances are presented in Figure 7.10b.



(a) Total tardiness. (b) Weighted total tardiness.

Figure 7.10: Performance comparison between `CPO` and `CPO+MDD` for minimizing total tardiness on randomly generated instances with 15 jobs.

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.

In all cases, a minimum width of 128 would suffice for the MDD propagation to provide enough inference to solve all the considered problems.

7.11 Conclusion

In this chapter we presented a novel generic approach to solving sequencing problems using multivalued decision diagrams (MDDs). We introduced relaxed MDDs to represent an over-approximation of all feasible solutions of a sequencing problem. We showed how these can be used to provide bounds on the objective function value or to derive structured sequencing information, such as precedence relations that must hold in any feasible solution. To strengthen the relaxed MDDs, we proposed a number of techniques for a large class of scheduling problems where precedence and time window constraints are imposed. We also showed that, for a TSP problem introduced by [10], the MDD that exactly represents all its feasible solutions has a polynomial size in the number of cities. Lastly, we have applied our MDD relaxations to constraint-based scheduling, and we showed that we can improve the performance of a state-of-the-art solver by orders of magnitude. In particular,

we were able to close three open TSPLIB instances for the TSP with precedence constraints. Relaxed MDDs thus provide a strong addition to existing generic approaches for solving constrained sequencing problems.

Chapter 8

Application: Timetabling

8.1 Introduction

In this chapter we will focus on particular timetabling applications formulated as constraint programming models [119, 46, 7]. The central inference process of constraint programming is *constraint propagation*. While traditional constraint processing techniques were designed for explicitly defined relations of small arity, state-of-the-art constraint programming solvers apply specialized constraint propagation algorithms for global constraints of any arity, often based on efficient combinatorial methods such as network flows [127, 114].

Conventional constraint propagation algorithms (or domain filtering algorithms) operate on individual constraints of a given problem. Their role is to identify and remove values in the variable domains that are inconsistent with respect to the constraint under consideration. Whenever the domain of a variable is updated (i.e., a value is removed), the constraints in which this variable appears can be reconsidered for inspection. This cascading process of propagating the changes in variable domains through the constraints continues until a fixed point is reached. Most constraint programming solvers assume that the variable domains are finite, which ensures termination of the constraint propagation process. Note that constraint propagation in itself may not be sufficient to determine the resolution of a given problem. Therefore, constraint propagation is normally applied at each search state in a systematic search process.

A major benefit of propagating variable domains is that it can be implemented efficiently in many cases. However, an inherent weakness of domain propagation is that it implicitly represents the Cartesian product of the variable domains as potential solution space. By communicating only domain changes, this limits the amount of information shared between constraints.

To address this shortcoming of domain propagation, [4] proposed the use of MDDs as an alternative to variable domains in the context of constraint propagation. MDDs can be used to represent individual (global) constraints, subsets of constraints, or all constraints in a given problem. When representing individual constraints, as in [81, 37], the higher-level information carried by the MDD

is lost when projecting this down to the variable domains for the traditional domain propagation. The highest potential for MDD propagation instead appears to be in representing specific subsets of constraints within the same MDD. That is, for a given set of constraints, we create and maintain one single limited-width MDD, which is then propagated through this constraint set. Since an MDD is defined with respect to a fixed variable ordering, it is most useful to select a subset of constraints compatible with this ordering. When applied in this way, MDD propagation can be implemented in parallel to the existing domain propagation in constraint programming systems, thus complementing and potentially strengthening the domain propagation process. For example, Chapter 7 introduced MDD propagation for a subset of constraints representing disjunctive scheduling problems. They embedded this as a custom global constraint in the ILOG CP Optimizer constraint programming solver, which greatly improved the performance.

Methodology

Constraint propagation based on limited-width MDDs amounts to applying *MDD filtering* and *MDD refinement* as described in Section 4.3. The role of an MDD filtering algorithm is to remove provably inconsistent arcs from the MDD [74, 84]. An MDD refinement algorithm on the other hand, aims at splitting nodes in the MDD to more accurately reflect the solution space [76]. In order to make this approach scalable and efficient, refinement algorithms must ensure that the MDD remains within a given maximum size (typically by restricting its maximum width—the number of nodes on any layer). By increasing this maximum width, the MDD relaxation can be strengthened to any desired level. That is, a maximum width of 1 would correspond to the traditional Cartesian product of the variable domains, while an infinite maximum width would correspond to an exact MDD representing all solutions. However, increasing the size of the MDD immediately impacts the computation time, and one typically needs to balance the trade-off between the strength of the MDD and the associated computation time.

In order to characterize the outcome of an MDD filtering algorithm, the notion of *MDD consistency* was introduced by [4], similar to domain consistency in finite-domain constraint programming: Given an MDD, a constraint is MDD consistent if all arcs in the MDD belong to at least one solution to the constraint. As a consequence of the richer data structure that an MDD represents, establishing MDD consistency may be more difficult than establishing domain consistency. For example, [4] show that establishing MDD consistency on the Alldifferent constraint is NP-hard, while establishing traditional domain consistency can be done in polynomial time [113].

Contributions

The main focus of this chapter is the Sequence constraint, that is defined as a specific conjunction of Among constraints, where an Among constraint restricts the occurrence of a set of values for a sequence of variables to be within a lower and upper bound [18]. The Sequence constraint finds

applications in, e.g., car sequencing and employee scheduling problems [115, 129]. It is known that classical domain consistency can be established for SEQUENCE in polynomial time [128, 129, 31, 101, 47]. Furthermore, [84] present an MDD filtering algorithm for AMONG constraints establishing MDD consistency in polynomial time. However, it remained an open question whether or not MDD consistency for SEQUENCE can be established in polynomial time as well.

In this work, we answer that question negatively and our first contribution is showing that establishing MDD consistency on the SEQUENCE constraint is NP-hard. This is an important 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 sub-constraints on contiguous variables, and existing approaches can handle this constraint fully by using bounds reasoning only.

As our second contribution, we show that establishing MDD consistency on the SEQUENCE constraint is fixed parameter tractable with respect to the lengths of the sub-sequences (the AMONG constraints), provided that the MDD follows the order of the SEQUENCE constraint. The proof is constructive, and follows from a generic 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 a new extension of MDD filtering to the information that is stored at its nodes.

Our last contribution is an experimental evaluation of our proposed partial MDD propagation algorithm. We evaluate the strength of our algorithm for MDDs of various maximum widths, and compare the performance with existing domain propagators for SEQUENCE. We also compare our algorithm with the currently best known MDD approach that uses the natural decomposition of SEQUENCE into AMONG constraints [84]. 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. Our results thus provide further evidence for the power of MDD propagation in the context of constraint programming.

The remainder of this chapter is structured as follows. In Section 8.2, we provide the necessary definitions of MDD-based constraint programming and the SEQUENCE constraint. In Section 8.3, we present the proof that establishing MDD consistency on SEQUENCE is NP-hard. Section 8.4 describes that establishing MDD consistency is fixed parameter tractable. In Section 8.5, the partial MDD filtering algorithm is presented. Section 8.6 shows the experimental results. We present final conclusions in Section 8.7.

8.2 Definitions

We first recall some basic definitions of MDD-based constraint programming, following [4, 84]. In this work, an *ordered Multivalued Decision Diagram (MDD)* is a directed acyclic graph whose nodes are partitioned into $n+1$ (possibly empty) subsets or *layers* $L_1, \ldots, L_{n+1}$, where the layers $L_1, \ldots, L_n$ correspond respectively to variables $x_1, \ldots, x_n$. L_1 contains a single *root* node r, and L_{n+1} contains a single *terminal* node t. For a node u in the MDD, we let Lu denote the index of its layer. For an MDD M, the *width* $w(M)$ is the maximum number of nodes in a layer, or $\max_{i=1}^{n}\{|L_i|\}$. In MDD-based CP, the MDDs typically have a given fixed maximum width.

All arcs of the MDD are directed from an upper to a lower layer; that is, from a node in some L_i to a node in some L_j with $i < j$. For our purposes it is convenient to assume (without loss of generality) that each arc connects two adjacent layers. Each arc out of layer L_i is labeled with an element of the domain $D(x_i)$ of x_i. For an arc a, we refer to the label it represents as $\ell(a)$. For notational convenience, we also write $\ell(u, v)$ instead of $\ell((u, v))$ for an arc (u, v). An element in $D(x_i)$ appears at most once as a label on the arcs out of a given node $u \in L_i$. The set $A(u, v)$ of arcs from node u to node v may contain multiple arcs, and we denote each with its label. Let $A^{in}(u)$ denote the set of arcs coming into node u. We define the size of an MDD M by the number of its arcs, i.e., $|M| = \big|\{a \mid a \in A^{in}(u), u \in L_i, i = 2, \ldots, n+1\}\big|$.

An arc with label v leaving a node in layer i represents an assignment $x_i = v$. Each path in the MDD from r to t can be denoted by the arc labels $v_1, \ldots, v_n$ on the path and is identified with the solution $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$. A path $v_1, \ldots, v_n$ is *feasible* for a given constraint C if setting $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$ satisfies C. Constraint C is feasible on an MDD if the MDD contains a feasible path for C.

A constraint C is called *MDD consistent* on a given MDD if every arc of the MDD lies on some feasible path. Thus MDD consistency is achieved when all redundant arcs (i.e., arcs on no feasible path) have been removed. We also say that such MDD is MDD consistent with respect to C. Domain consistency for C is equivalent to MDD consistency on an MDD of width one that represents the variable domains. That is, it is equivalent to MDD consistency on an MDD in which each layer L_i contains a single node s_i, and $A(s_i, s_{i+1}) = D(x_i)$ for $i = 1, \ldots, n$.

Lastly, we formally recall the definitions of Among [18], Sequence [18], and Gen-Sequence [129] constraints. 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:

Definition 34 *Let X be a set of variables, l, u integer numbers such that $0 \le l \le u \le |X|$, and $S \subset \cup_{x \in X} D(x)$ a subset of domain values. Then we define* $\textsc{Among}(X, l, u, S)$ *as*

$$l \le \sum_{x \in X} (x \in S) \le u.$$

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 SEQUENCE constraint is the conjunction of a given AMONG constraint applied to every sub-sequence of length q over a sequence of n variables:

Definition 35 *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*

$$\text{SEQUENCE}(X, q, l, u, S) = \bigwedge_{i=1}^{n-q+1} \text{AMONG}(s_i, l, u, S),$$

where s_i represents the sub-sequence $x_i, \ldots, x_{i+q-1}$.

Finally, the *generalized* SEQUENCE constraint extends the SEQUENCE constraint by allowing the AMONG constraints to be specified with different lower and upper bounds, and sub-sequence length:

Definition 36 *Let X be an ordered set of n variables, k a natural number, $\vec{s}$, $\vec{l}$, $\vec{u}$ vectors of length k such that s_i is a sub-sequence 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*

$$\text{GEN-SEQUENCE}(X, \vec{s}, \vec{l}, \vec{u}, S) = \bigwedge_{i=1}^{k} \text{AMONG}(s_i, l_i, u_i, S).$$

8.3 MDD Consistency for Sequence is NP-Hard

As stated before, the only known non-trivial NP-hardness result for a global constraint in the context of MDD-based constraint programming is that of [4] for the ALLDIFFERENT constraint. 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. For SEQUENCE constraints, so far it was unknown whether a polynomial-time MDD consistency algorithm exists. In this section we answer that question negatively and prove the following result.

THEOREM 37 *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 [63]. 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.

Consider a 3-SAT instance on n variables $x_1, \ldots, x_n$, consisting of m clauses $c_1, \ldots, c_m$. We first construct an MDD that represents the basic structure of the 3-SAT formula (see Example 23 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 $i = 1, \ldots, m$ and $j = 1, \ldots, n$ (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 $Y = y_{1,1}, \overline{y}_{1,1}$, $y_{1,2}, \overline{y}_{1,2}, \ldots, y_{1,n}, \overline{y}_{1,n}, \ldots, y_{m,1}, \overline{y}_{m,1}, \ldots, y_{m,n}, \overline{y}_{m,n}$. 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 $y_{i,1}, \ldots, \overline{y}_{i,n}$. 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 $Ly_{i,1}$, and we set $tag(r_i) =$ 'unsat'. For each node u in layer $Ly_{i,j}$ (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 $L\overline{y}_{i,j}$, one single node w in $Ly_{i,j+1}$, 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 $L\overline{y}_{i,j}$, two nodes w, w' in $Ly_{i,j+1}$, 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 Ly_{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 $Ly_{i,1}$ and $Ly_{i+1,1}$ 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

$$\textsc{Sequence}(Y, q = 2n, l = n, u = n, S = \{1\}) \tag{8.1}$$

on the MDD M described above. If the sub-sequence of length $2n$ starts from a positive literal $y_{i,j}$, by definition there are exactly n variables that take value 1. If the sub-sequence starts from a

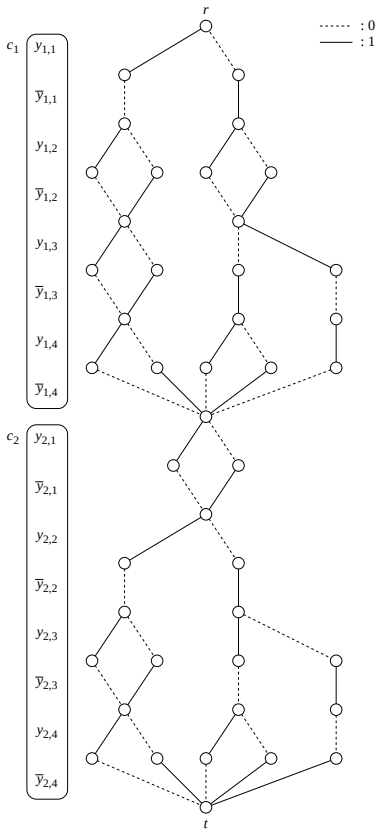


Figure 8.1: The MDD corresponding to Example 23.

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 sub-sequences, all variables x_j must take the same value in all clauses.

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 23 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 Figure 8.1. □

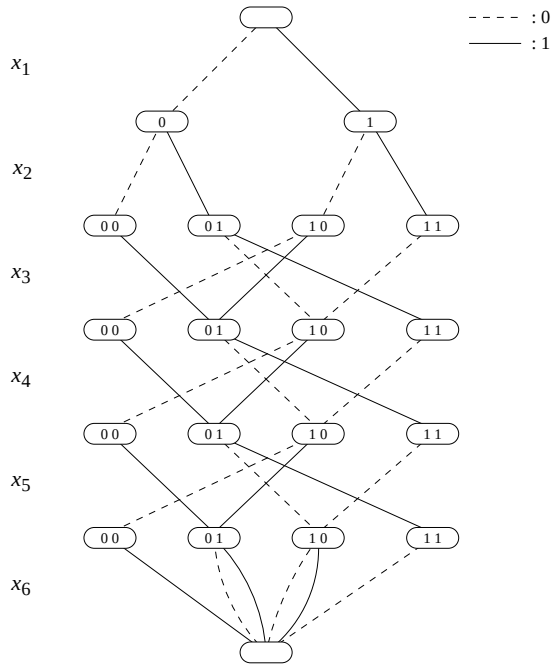


Figure 8.2: The exact MDD for the SEQUENCE constraint of Example 24.

8.4 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 of the sub-sequences q. It was already shown in [128, 129] 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 24 Consider the constraint SEQUENCE$(X, q = 3, l = 1, u = 2, S = \{1\})$ 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 Figure 8.2. For convenience, each node in the MDD is labeled with the last $q-1$ labels that represent the sub-sequence 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 sub-sequence 01. To construct the next layer, we either append a 0 or a 1 to this sub-sequence (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 sub-sequences have been created (maximally $O(2^{q-1})$), which thus defines the width of the subsequent layers. □
However, since we are given an arbitrary MDD, and not necessarily an exact MDD, we need some additional steps to exploit this connection. For this we apply a generic approach that will not only

show fixed parameter tractability for SEQUENCE, but in fact can be applied to determine whether MDD consistency is tractable for any constraint.

Our goal is to establish MDD consistency on a given MDD M with respect to another MDD M' on the same set of variables. This is compatible with our earlier definitions since M' can be interpreted to define a constraint. That is, M is MDD consistent with respect to M' if every arc in M belongs to a path (solution) that also exists in M'. For our purposes, we assume that M and M' follow the same variable ordering.

We can establish 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 [94].

The intersection MDD, denoted by I, represents all possible paths (solutions) that are present both in 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 re-use 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 traversal, 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 38 *Algorithm 8 establishes MDD-consistency on M with respect to M' in $O(|M| \cdot w(M')$ time and space.*

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

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 $s(u)$.

create node r^I with state $s(r^I) := (r, r')$
$L_1^I := \{r^I\}$
for $i = 1$ **to** n **do**
  $L_{i+1}^I := \{\}$
  for all $u \in L_i^I$ with $s(u) = (v, v')$ **do**
    for all $a = (v, \tilde{v}) \in M$ and $a' = (v', \tilde{v}') \in M'$ such that $\ell(a) = \ell(a')$ **do**
      create node $\tilde{u}$ with state $s(\tilde{u}) := (\tilde{v}, \tilde{v}')$
      if $\exists\, \tilde{w} \in L_{j+1}^I$ with $s(\tilde{w}) = s(\tilde{u})$ **then** $\tilde{u} := \tilde{w}$
      else $L_{i+1}^I \mathrel{+}= \tilde{u}$ **end if**
      add arc $(u, \tilde{u})$ with label $\ell(a)$ to arc set A^I
remove all arcs and nodes from I that are not on a path from r^I to $t^I \in L_{n+1}^I$
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'

create I := Intersection(M,M')
for $i = 1$ **to** n **do**
  create array Support$[u, l] := 0$ for all $u \in L_i^M$ and arcs out of u with label l
  for all arcs $a = (v, \tilde{v})$ in A^I with $s(v) = (u, u')$ such that $v \in L_i^I$ **do**
    Support$[u, \ell(a)] := 1$
  for all arcs $a = (u, \tilde{u})$ in M such that $u \in L_i^M$ **do**
    if Support$[u, \ell(a)] = 0$ **then** remove a from M **end if**
remove all arcs and nodes from M that are not on a path from r to $t \in L_{n+1}^M$
return M

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), $w(M) \cdot w(M')$ (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 $w(M')$ 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 $O(|M| \cdot w(M'))$. The memory requirements are bounded by the size of the intersection, which is at most $O(n \cdot w(M) \cdot w(M') \cdot d_{\max}) = O(|M| \cdot w(M'))$. 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.

We next apply Theorem 38 to the SEQUENCE constraint.

COROLLARY 39 *Let X be an ordered sequence of variables, $C = \text{SEQUENCE}(X, q, l, u, S)$ a sequence constraint, and M 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 [128, 129] that there exists an exact MDD M' of size $O(n2^{q-1})$ that represents C. Applying Theorem 38 gives an MDD-consistency algorithm with time and space complexity $O(|M|\, 2^{q-1})$, and the result follows. ■

We note that Theorem 38 can also be applied to obtain the tractability of establishing MDD consistency on other constraints. Consider for example the constraint $\text{AMONG}(x_1, x_2, \ldots, x_n, l, u, S)$. For any variable ordering, we can construct an exact MDD in a top-down procedure by associating with each node v the number of variables taking a value in S along the path from r to v, representing the 'length' of that path. Nodes with the same length are equivalent and can be merged. Because the largest layer has at most $u + 1$ different path lengths, the exact MDD has size $O(nu)$, and by

Theorem 38 establishing MDD consistency is tractable for AMONG. Indeed, [84] also showed that MDD consistency can be established for this constraint, with quadratic time complexity.

The converse of Theorem 38 does not hold: there exist constraints for which MDD consistency can be established 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 established 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]. However, [88] provide the following explicit linear inequality. 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 $1 \leq i, j \leq k$. They show that, for any variable order, the size of the reduced ordered BDD for this inequality is bounded from below by $\Omega(2^{\sqrt{n}/2})$.

8.5 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 39. 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 [84]. 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 [128, 129, 31, 101]. 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.

8.5.1 Cumulative Sums Encoding

Our proposed algorithm extends the original domain consistency filtering algorithm for SEQUENCE by [128] to MDDs, following the 'cumulative sums' encoding as proposed by [31]. This representation takes the following form. For a sequence of variables $X = x_1, x_2, \ldots, x_n$, and a constraint SEQUENCE(X, q, l, u, S), 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:

$$y_i = y_{i-1} + \delta_S(x_i) \qquad \forall i \in \{1, \ldots, n\}, \tag{8.2}$$

$$y_{i+q} - y_i \geq l \qquad \forall i \in \{0, \ldots, n-q\}, \tag{8.3}$$

$$y_{i+q} - y_i \leq u \qquad \forall i \in \{0, \ldots, n-q\}, \tag{8.4}$$

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 $x \notin S$. [31] 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 $[\mathrm{lb}(v), \mathrm{ub}(v)]$, and we will refer to it as the 'node domain' of v. Following the approach of [84], we can compute this information in linear time by one top-down pass, by using equation (8.2), as follows:

$$\begin{aligned} \mathrm{lb}(v) &= \min_{(u,v)\in A^{in}(v)} \left\{\mathrm{lb}(u) + \delta_S\left(\ell(u,v)\right)\right\}, \\ \mathrm{ub}(v) &= \max_{(u,v)\in A^{in}(v)} \left\{\mathrm{ub}(u) + \delta_S\left(\ell(u,v)\right)\right\}, \end{aligned} \tag{8.5}$$

for all nodes $v \neq r$, while $[\mathrm{lb}(r), \mathrm{ub}(r)] = [0, 0]$.

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 $\mathcal{A}_v$ of length $q+1$ for each node v, where $\mathcal{A}_v[i]$ represents the set of ancestor nodes of v at the i-th layer above v, for $i = 0, \ldots, q$. We initialize $\mathcal{A}_r = [\{r\}, \emptyset, \ldots, \emptyset]$, and apply the recursion

$$\begin{aligned} &\mathcal{A}_v[i] = \cup_{(u,v)\in A^{in}(v)} \mathcal{A}_u[i-1] \qquad \text{for } i = 1, 2, \ldots, q, \\ &\mathcal{A}_v[0] = \{v\}. \end{aligned}$$

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 $O(q \cdot (w(M))^2)$ operations for an MDD M. Now, the relevant ancestor nodes for a node v in layer L_{i+q} are stored in $\mathcal{A}_v[q]$, a subset of layer L_i. We similarly compute all descendant nodes of v in a vector $\mathcal{D}_v$ of length $q+1$, such that $\mathcal{D}_v[i]$ contains all descendants of v in the i-th layer below v, for $i = 0, 1, \ldots, q$. We initialize $\mathcal{D}_t = [\{t\}, \emptyset, \ldots, \emptyset]$.

However, for our purposes we only need to maintain the minimum and maximum value of the union of the domains of $\mathcal{A}_v$, resp., $\mathcal{D}_v$, because constraints (8.3) and (8.4) are inequalities; see the application of $\mathcal{A}_v$ and $\mathcal{D}_v$ in rules (8.8) below. This makes the recursive step more efficient, now taking $O(qw(M))$ operations per node.

Alternatively, we can approximate this information by only maintaining a minimum and max-

imum 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.

8.5.2 Processing the Constraints

We next process each of the constraints (8.2), (8.3), and (8.4) in turn to remove provably inconsistent arcs, while at the same time we filter the node information.

Starting with the ternary constraints of type (8.2), we remove an arc (u, v) if $\mathrm{lb}(u)+\delta_S(\ell(u, v)) > ub(v)$. Updating $[\mathrm{lb}(v), \mathrm{ub}(v)]$ for a node v is done similar to the rules (8.5) above:

$$\begin{aligned} \mathrm{lb}(v) &= \max\left\{\mathrm{lb}(v), \min_{(u,v)\in A^{in}(v)}\left\{\mathrm{lb}(u) + \delta_S(\ell(u, v))\right\}\right\}, \\ \mathrm{ub}(v) &= \min\left\{\mathrm{ub}(v), \min_{(u,v)\in A^{in}(v)}\left\{\mathrm{ub}(u) + \delta_S(\ell(u, v))\right\}\right\}, \end{aligned} \tag{8.6}$$

In fact, the resulting algorithm is a special case of the MDD consistency equality propagator of [76], and we thus inherit the MDD consistency for our ternary constraints.

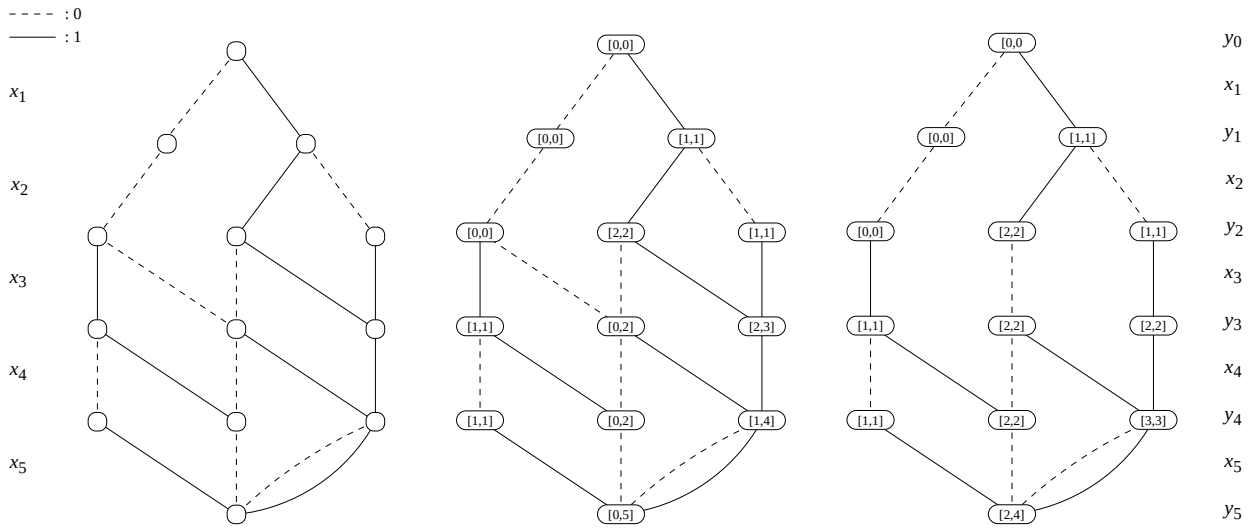
Next, we process the constraints (8.3) and (8.4) for a node v in layer L_{i+1} ($i = 0, \ldots, n$). Recall that the relevant ancestors from L_{i+1-q} are $\mathcal{A}_v[q]$, while its relevant descendants from L_{i+1+q} are $\mathcal{D}_v[q]$. The variable corresponding to node v is y_i, and it participates in four constraints:

$$\begin{aligned} y_i &\geq l + y_{i-q}, \\ y_i &\leq u + y_{i-q}, \\ y_i &\leq y_{i+q} - l, \\ y_i &\geq y_{i+q} - u. \end{aligned} \tag{8.7}$$

Observe 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 (8.7):

$$\begin{aligned} \mathrm{lb}(v) = \max\{ &\quad \mathrm{lb}(v), &\quad l + \min_{u\in\mathcal{A}_v[q]} \mathrm{lb}(u), &\quad \min_{w\in\mathcal{D}_v[q]} \mathrm{lb}(w) - u \quad \}, \\ \mathrm{ub}(v) = \min\{ &\quad \mathrm{ub}(v), &\quad u + \max_{u\in\mathcal{A}_v[q]} \mathrm{ub}(u), &\quad \max_{w\in\mathcal{D}_v[q]} \mathrm{ub}(w) - l \quad \}. \end{aligned} \tag{8.8}$$

The resulting algorithm is a specific instance of the generic MDD consistent binary constraint propagator presented by [84], 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.



a. Initial MDD b. Node domains c. MDD after filtering

Figure 8.3: MDD propagation for the constraint SEQUENCE($X, q = 3, l = 1, u = 2, S = \{1\}$) of Example 25.

EXAMPLE 25 Consider the constraint SEQUENCE($X, q = 3, l = 1, u = 2, S = \{1\}$) 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 Figure 8.3.a. In Figure 8.3.b. we show the node domains that result from processing rules (8.5). Figure 8.3.c. shows the resulting MDD after processing the constraints via the rules (8.6) and (8.8). 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 $\mathcal{A}_v[q]$ 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 Figure 8.3.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 $(1, 1, 0, 0, 0)$. □

Observe that our proposed algorithm can be applied immediately to the more general GEN-SEQUENCE constraints in which each AMONG constraint has its individual l, u and q. The cumulative sums encoding can be adjusted in a straightforward manner to represent these different values.

8.5.3 Formal Analysis

We next formally compare the outcome of our partial MDD filtering algorithm with MDD propagation for the AMONG encoding and domain propagation for SEQUENCE. First, we recall Theorem 4 from [31].

THEOREM 40 **[31]** *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.

COROLLARY 41 *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 [31]. Consider the constraint SEQUENCE$(X, q = 2, l = 1, u = 2, S = \{1\})$ with the ordered sequence of binary variables $X = \{x_1, x_2, x_3, x_4\}$ having 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

$$y_0 \in [0, 0], y_1 \in [0, 1], y_2 \in [1, 2], y_3 \in [1, 2], y_4 \in [2, 3],$$
$$x_1 \in \{0, 1\}, x_2 \in \{0, 1\}, x_3 \in \{0\}, x_4 \in \{0, 1\}.$$

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 SEQUENCE$(X, q = 3, l = 1, u = 1, S = \{1\})$ with the ordered sequence of binary variables $X = \{x_1, x_2, x_3, x_4\}$ having 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

$$y_0 \in [0, 0], y_1 \in [0, 0], y_2 \in [0, 1], y_3 \in [1, 1], y_4 \in [1, 1],$$
$$x_1 \in \{0\}, x_2 \in \{0, 1\}, x_3 \in \{0, 1\}, \mathbf{x_4} \in \{\mathbf{0}\},$$

while establishing MDD consistency on the AMONG encoding does not prune any value. ■

As an additional illustration of Corollary 41, consider again Example 25 and Figure 8.3. 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 42 *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 41 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 $\text{SEQUENCE}(X, q, l, u, S = \{1\})$ 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 to these other methods.

8.6 Computational Results

The purpose of our computational results is to evaluate empirically the strength of the partial MDD propagator described in Section 8.5. 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 to the filtering strength of the MDD propagators for the individual AMONG constraints, being the best MDD approach for SEQUENCE so far [84].

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 8.5 that for applying rules (8.8) 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 [84]. For the domain propagation, we applied three models. The first uses the domain consistent propagator for SEQUENCE from [129], running in $O(n^3)$ time. The second uses the domain consistent propagator for SEQUENCE based

on a network flow representation by [101], which runs in $O(n^2)$ time.[1] As 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 [31] 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). [107] shows that establishing bounds consistency on such 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 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 2.33GHz Intel Xeon machine.

8.6.1 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 5 SEQUENCE constraints. For each SEQUENCE constraint, we set the length of sub-sequence 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

[1] We thank Nina Narodytska for sharing the implementation with us.

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 first 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.

We compare the performance of domain propagation and MDD propagation in Figure 8.4. 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

[2]All instances are available at `http://www.andrew.cmu.edu/user/vanhoeve/mdd/`.

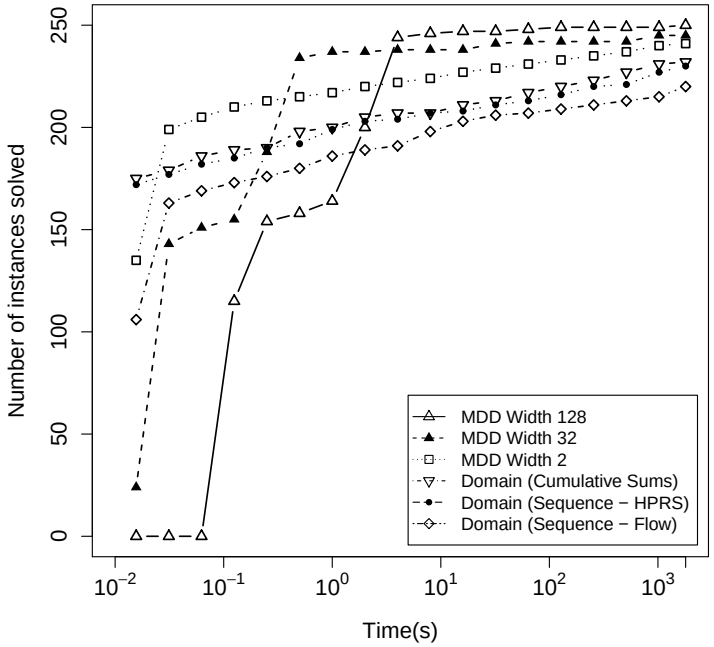
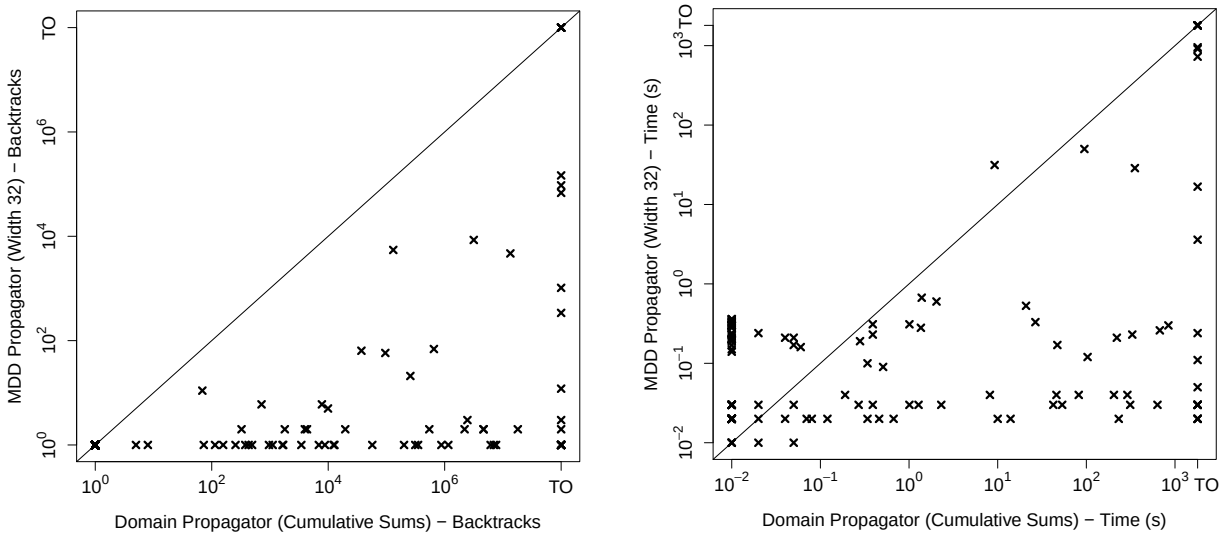


Figure 8.4: Performance comparison of domain and MDD propagators for the Sequence constraint. Each data point reflects the total number of instances that are solved by a particular method within the corresponding time limit.



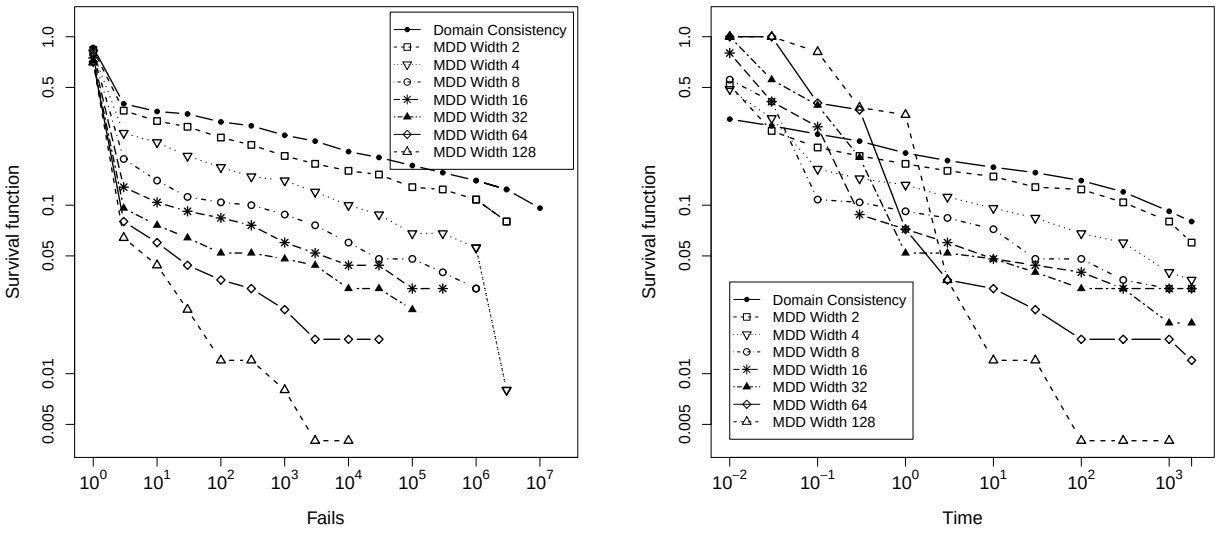
a. Number of backtracks

b. Solving time

Figure 8.5: 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 1,800s (Fig. b.). In Figure 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.

[128, 129]), and 'Sequence - Flow' (the flow-based propagator by [101]). 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 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, Figure 8.5 presents scatter plots comparing domain propagation (cumulative 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, Figure 8.5.a depicts the number of backtracks while Figure 8.5.b depicts the solving time of both methods. The instances that



a. Survival function with respect to backtracks

b. Survival function with respect to solving time

Figure 8.6: 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.

were not solved within the time limit are collected under 'TO' (time out) for that method. Figure 8.5.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 Figure 8.5.b shows that for almost all instances (especially the harder ones), the search tree reductions correspond to faster solving times, again often several orders of magnitude.

We next evaluate the impact of increasing maximum widths of the MDD propagator. In Figure 8.6, 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 [67]. In our case, we approximate this function by taking the proportion of instances that need at least x backtracks (Figure 8.6.a), respectively seconds (Figure 8.6.b). Observe that these are log-log plots. With respect to the search tree size, Figure 8.6.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 Figure 8.6.b. It reflects similar behavior, but also takes into account the initial computational overhead of MDD propagation.

Requirement	SEQUENCE(X, q, l, u, S)
At least 20 work shifts every 28 days:	SEQUENCE$(X, 28, 20, 28, \{D, E, N\})$
At least 4 off-days every 14 days:	SEQUENCE$(X, 14, 4, 14, \{O\})$
Between 1 and 4 night shifts every 14 days:	SEQUENCE$(X, 14, 1, 4, \{N\})$
Between 4 and 8 evening shifts every 14 days:	SEQUENCE$(X, 14, 4, 8, \{E\})$
Nights shifts cannot appear on consecutive days:	SEQUENCE$(X, 2, 0, 1, \{N\})$
Between 2 and 4 evening/night shifts every 7 days:	SEQUENCE$(X, 7, 2, 4, \{E, N\})$
At most 6 work shifts every 7 days:	SEQUENCE$(X, 7, 0, 6, \{D, E, N\})$

Table 8.1: 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).

	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

Table 8.2: 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.

8.6.2 Nurse Rostering Instances

We next consider a more structured problem class inspired by nurse rostering problems. The problem is to design a work 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. We impose the eight SEQUENCE constraints modeling the requirements listed in Table 8.1.

By the combinatorial nature of this problem, the size of the CP search tree turns out to be largely independent on the 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 8.2. 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

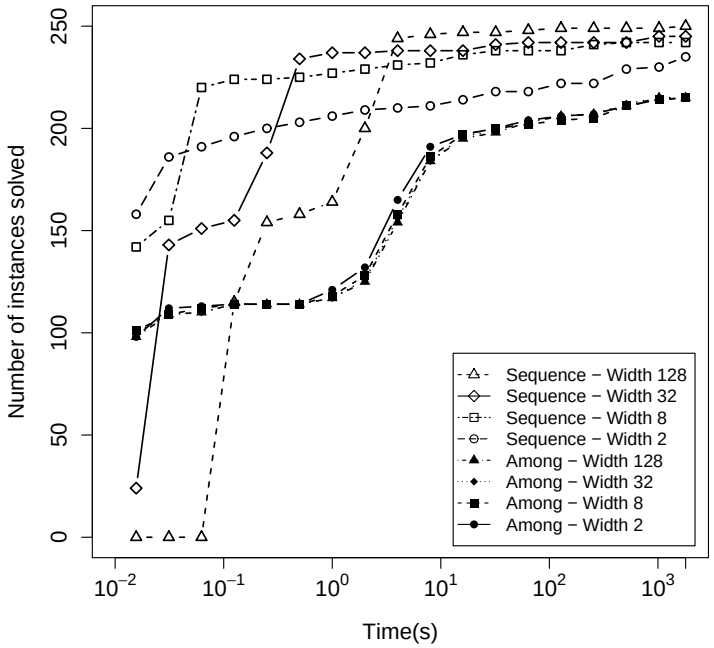


Figure 8.7: Performance comparison of MDD propagation for SEQUENCE and AMONG for various maximum widths. Each data point reflects the total number of instances that are solved by a particular method within the corresponding time limit.

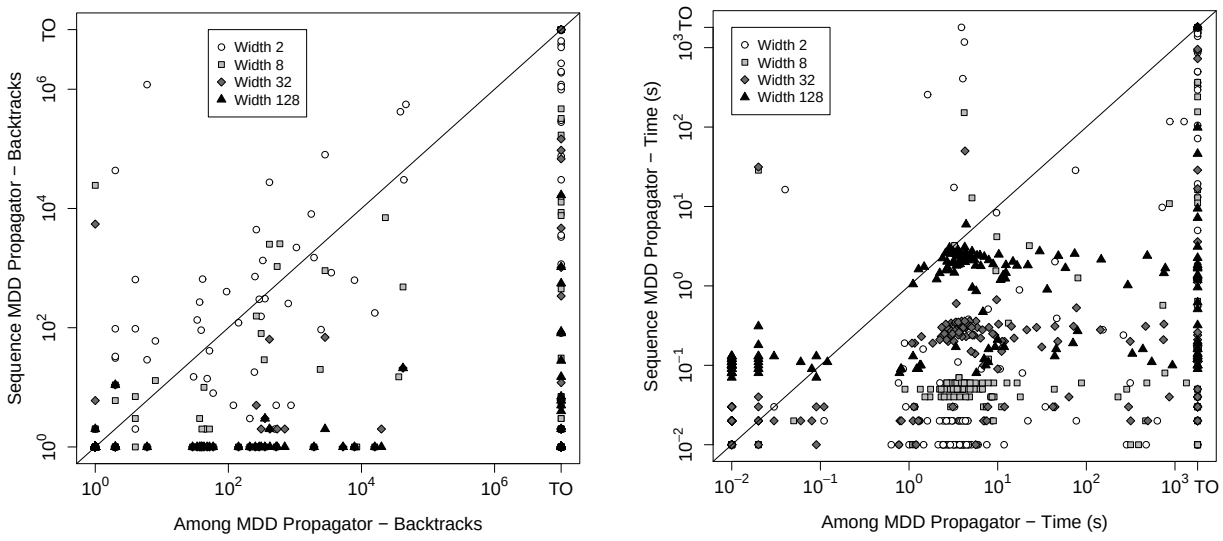
equivalent to the domain propagator of the cumulative sums. As a first observation, a maximum width of 2 already reduces the number of backtracks by a factor 8.3. For maximum width of 8 the MDD propagation even allows to solve the problem without search. The computation times are correspondingly reduced, e.g., from 157s (resp. 96s) for the domain propagators to 0.10s 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 to 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 others instances were solved within the time limit of 1,800 seconds.

8.6.3 Comparing MDD Filtering for Sequence and Among

In our last experiment, we compare our SEQUENCE MDD propagator to the MDD propagator for AMONG constraints by [84]. 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, MDD propagation for SEQUENCE and MDD propagation for AMONG, to the data set of Section 8.6.1 containing 250 instances. The time limit is again 1,800 seconds,



a. Number of backtracks

b. Solving time

Figure 8.8: 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 Figure 8.5.

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 Figure 8.7. 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 propagator with maximum width 2 outperforms the Among propagators for all maximum widths up to 128.

The scatter plot in Figure 8.8 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 Figure 8.8.a, resp. 8.8.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 to the Among decomposition.

8.7 Conclusion

Constraint propagation with limited-width MDDs has recently been shown to be a powerful alternative to the conventional propagation of variable domains in constraint programming. In this work, we have studied MDD propagation for the Sequence constraint, which appears in, e.g., rostering and scheduling applications. We have first proved that establishing MDD consistency for Sequence is NP-hard. However, we have also shown that this task is fixed parameter tractable with respect to the length of the sub-sequences defined by the constraint, provided that the MDD follows the variable ordering specified by the constraint. We then proposed a practical MDD propagation algorithm for Sequence that is also polynomial in the length of the sub-sequences, which is based on a cumulative decomposition. We provided extensive experimental results comparing our MDD propagator for Sequence to domain propagators for Sequence as well as an existing MDD propagator for Among. Our computational experiments have shown that our MDD propagator for Sequence can outperform domain propagators by orders by magnitude in terms of search tree size and solving time. Similar results were obtained when compared to the existing MDD propagator for Among, which demonstrates that in practice a large MDD alone is not sufficient to solve these problems; specific MDD propagators for global constraints such as Sequence can lead to orders of magnitude speedups.

Chapter 9

Conclusion

The main objective of this dissertation is to enhance the modeling and solving capabilities of generic optimization technology through the use of decision diagrams (DDs). To this end, we extended the analysis and application of decision diagrams for optimization problems in several ways.

First, we provided a modeling framework based on dynamic programming that can be used to specify how to build a decision diagram of a discrete optimization problem and how to relax it, facilitating the encoding process of a problem to a diagram representation. Using a number of classical optimization problems as test cases, we analyzed the strength of the bounds obtained from a relaxed decision diagram and those provided by a *restricted decision diagram*. This is a new type of limited-size diagram introduced by us that only encodes feasible solutions of the problem, but not necessarily all of them, thus defining a novel generic primal heuristic. We observed that both relaxed and restricted diagrams were superior to state-of-the-art integer programming technology when a large number of variables was involved.

We then introduced a novel branch-and-bound technique based on relaxed and restricted decision diagrams. The key idea of the technique is to branch on the *nodes* of a relaxed decision diagram, which eliminates symmetry in search as they aggregate partial assignments belonging to the same equivalence class. Computational experiments for different problem classes indicate that our rudimentary implementation is competitive with state-of-the-art generic optimization technology. In particular, we were able to reduce the optimality gap of benchmark maximum cut instances that still remain unsolved.

Another key characteristic of our branch-and-bound method is that it can be easily parallelized and does not require too much communication among computer cores or complex load balancing heuristics. It could therefore be suitable to cloud computing or to clusters with hundreds of computer cores. We performed experiments on the maximum independent set problem, and obtained almost linear solving time speed-ups on a cluster with 256 cores.

We also studied the application of relaxed diagrams to specific domains; namely, to *sequencing problems* and to *timetabling* problems. We showed that our approach can be embedded into a state-

of-the-art constraint programming solver to speed-up solving times by orders of magnitude. We were specifically able to close open TSPLib benchmark instances of a traveling salesman problem version where precedence constraints must be considered.

In summary, the resulting theoretical and empirical study of the techniques in this thesis indicate four favorable characteristics of decision diagrams in the context of optimization:

- *Modeling flexibility:* The use of decision diagrams permits a more flexible approach to modeling, since no particular structure is imposed on the constraints or objective function of the problem. Instead, it only requires the problem to be formulated in a recursive way. This may be more appropriate, e.g., for problems with no adequate integer linear formulation, such as the maximum cut problem.

- *Potentially suitable to large-scale problems:* All involved operations are performed in computationally inexpensive steps over a diagram with a parameterized size. We observed that relevant information, such as non-trivial bounds, can be obtained even when the imposed decision diagram sizes are small. Large scale problems are thus potential candidates to be tackled with our techniques, for instance by adjusting the diagram size to take into account the available computing resources.

- *Alternative inference technique:* Besides the computation of bounds, relaxed decision diagram can also be used for *inference* purposes, i.e. to deduce new constraints of a problem. This inference provides a method to link decision diagrams with other optimization techniques, since the deduced constraints may be applied, e.g., to strengthen an integer programming formulation of the problem or to derive new filtering algorithms in constraint programming. For example, we showed that a relaxed decision diagram for certain scheduling problems allows us to deduce non-trivial precedence relations between jobs, which can be incorporated in constraint-based schedulers to significantly speed-up search.

- *Parallelization:* The recursive structure of a decision diagram permits more natural parallelization strategies suitable for thousands of computer cores. Moreover, relaxed decision diagrams can also be perceived as an approximation of a branching tree, providing measures that can be used in load balancing heuristics, for example.

Decision diagrams provide a fresh perspective in optimization and many research directions are possible in the near future. For instance, one could study different diagram representations for integer programming models and how they could be relaxed or restricted. Moreover, one may also study the relationship between a relaxed decision diagram and other forms of relaxation, such as a linear programming relaxation or an approximate dynamic programming model. In summary, the range of applications is quite exciting and can lead to significant insights and enhancements on the way how discrete optimization problems are modeled and solved.

Bibliography

[1] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, New York, 1997.

[2] Hernn Abeledo, Ricardo Fukasawa, Artur Pessoa, and Eduardo 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. Hadzic, J. N. Hooker, and P. Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.

[5] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, pages 118–132, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Davide Anghinolfi, Roberto Montemanni, Massimo Paolucci, and Luca Maria 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] Norbert Ascheuer. *Hamiltonian Path Problems in the On-line Optimization of Flexible Manufacturing Systems*. PhD thesis, Technische Universitt Berlin, Germany, 1995.

[9] Norbert Ascheuer, Michael Jnger, and Gerhard Reinelt. A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17:2000, 2000.

[10] E. Balas. New classes of efficiently solvable generalized traveling salesman problems. *Annals of Operations Research*, 86:529–558, 1999.

[11] Egon Balas. A linear characterization of permutation vectors. Management science research report 364, Carnegie Mellon University, 1975.

[12] Egon Balas and Neil Simonetti. Linear time dynamic-programming algorithms for new classes of restricted tsps: A computational study. *INFORMS J. on Computing*, 13:56–75, December 2000.

[13] 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.

[14] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research and Management Science. Kluwer, 2001.

[15] B. Becker, M. Behle, F. Eisenbrand, and R. Wimmer. BDDs in a branch and cut framework. In S. Nikoletseas, editor, *Experimental and Efficient Algorithms, Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA 05)*, volume 3503 of *Lecture Notes in Computer Science*, pages 452–463. Springer, 2005.

[16] M. Behle. *Binary Decision Diagrams and Integer Programming*. PhD thesis, Max Planck Institute for Computer Science, 2007.

[17] M. Behle and F. Eisenbrand. 0/1 vertex and facet enumeration with BDDs. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 158–165, 2007.

[18] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[19] D. Bergman, W.-J. van Hoeve, and J. N. Hooker. Manipulating MDD Relaxations for Combinatorial Optimization. In *Proceedings of CPAIOR*. Springer, 2011. To appear.

[20] David Bergman. *New Techniques for Discrete Optimization*. PhD thesis, Tepper School of Business, Carnegie Mellon University, 2013.

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

[22] David Bergman, André A. Ciré, and Willem Jan van Hoeve. Mdd propagation for sequence constraints. *J. Artif. Intell. Res. (JAIR)*, 50:697–722, 2014.

[23] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John N. Hooker. Variable ordering for the application of bdds to the maximum independent set problem. In *Proceedings*

of the 9th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR'12, pages 34–49, Berlin, Heidelberg, 2012. Springer-Verlag.

[24] David Bergman, AndreA. Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat, and Willem-Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, volume 8451 of *Lecture Notes in Computer Science*, pages 351–367. Springer International Publishing, 2014.

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

[26] David Bergman, Willem-Jan van Hoeve, and John Hooker. Manipulating MDD relaxations for combinatorial optimization. In Tobias Achterberg and J. Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6697 of *Lecture Notes in Computer Science*, pages 20–35. Springer Berlin / Heidelberg, 2011.

[27] Timo Berthold. Primal heuristics for mixed integer programs. Master's thesis, Zuze Institute Berlin, 2006.

[28] Dimitris Bertsimas, Dan A. Iancu, and Dmitriy Katz. A new local search algorithm for binary optimization. *INFORMS Journal on Computing*, 25(2):208–221, 2013.

[29] B. Bloom, D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. Saraswat. SatX10: A scalable plug & play parallel SAT framework. In *Proceedings of SAT*, volume 7317 of *LNCS*, pages 463–468. Springer, 2012.

[30] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45:993–1002, 1996.

[31] S. Brand, N. Narodytska, C.G. Quimper, P. Stuckey, and T. Walsh. Encodings of the Sequence Constraint. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 210–224. Springer, 2007.

[32] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[33] Neil J. Calkin and Herbert S. Wilf. The number of independent sets in a grid graph. *SIAM J. Discrete Math.*, 11(1):54–60, 1998.

[34] Alberto Caprara, Matteo Fischetti, and Paolo Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:2000, 1998.

[35] S. Chang, Q. Lu, G. Tang, and W. Yu. On decomposition of the total tardiness problem. *Operations Research Letters*, 17(5):221 – 229, 1995.

[36] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, San Diego, CA, USA, 2005.

[37] K. Cheng and R. Yap. Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints. In *Proceedings of CP*, volume 5202 of *LNCS*, pages 509–523. Springer, 2008.

[38] Marielle Christiansen and Kjetil Fagerholt. Maritime inventory routing problems maritime inventory routing problems. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 1947–1955. Springer US, 2009.

[39] Nicos Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

[40] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-Based Work Stealing in Parallel Constraint Programming. In *Proceedings of CP*, pages 226–241, 2009.

[41] A. A. Cire and W.-J. van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61(6):1411–1428, 2013.

[42] Andre A. Cire and John Hooker. The separation problem for decision diagrams. In *ISAIM*, 2014.

[43] Andre A. Cire and Willem-Jan van Hoeve. MDD propagation for disjunctive scheduling. In *Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS*, pages 11–19. AAAI Press, 2012.

[44] Gianna M. Del Corso and Giovanni Manzini. Finding exact solutions to the bandwidth minimization problem. *Computing*, 62(3):189–203, 1999.

[45] James Cussens. Bayesian network learning by compiling to weighted max-sat. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence (UAI 2008)*, pages 105–112, Helsink, 2008.

[46] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[47] N. Downing, T. Feydy, and P. Stuckey. Explaining Flow-Based Propagation. In *Proceedings of CPAIOR*, volume 7298 of *LNCS*, pages 146–162. Springer, 2012.

[48] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.

[49] Martin E. Dyer, Alan M. Frieze, and Mark Jerrum. On counting independent sets in sparse graphs. *SIAM J. Comput.*, 31(5):1527–1541, 2002.

[50] R. Ebendt, W. Gunther, and R. Drechsler. An improved branch and bound algorithm for exact BDD minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(12):1657–1663, 2003.

[51] J. D. Eblen, C. A. Phillips, G. L. Rogers, and M. A. Langston. The maximum clique enumeration problem: Algorithms, applications and implementations. In *Proceedings of the 7th international conference on Bioinformatics research and applications*, ISBRA'11, pages 306–319, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] Jonathan Eckstein and Mikhail Nediak. Pivot, cut, and dive: a heuristic for 0-1 mixed integer programming. *J. Heuristics*, 13(5):471–503, 2007.

[53] 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-Verlag, 1999.

[54] Uriel Feige. Approximating the bandwidth via volume respecting embeddings. *J. Comput. Syst. Sci*, 60(3):510–539, 2000.

[55] 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.

[56] Samuel Fiorini, Serge Massar, Sebastian Pokutta, Hans Raj Tiwary, and Ronald de Wolf. Linear vs. semidefinite extended formulations: Exponential separation and strong lower bounds. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 95–106, New York, NY, USA, 2012. ACM.

[57] Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Math. Program*, 104(1):91–104, 2005.

[58] Matteo Fischetti and Domenico Salvagnin. Feasibility pump 2.0. *Mathematical Programming Computation*, 1(2-3):201–222, 2009.

[59] Florence Forbes and Bernard Ycart. Counting stable sets on cartesian products of graphs. *Discrete Mathematics*, 186(1-3):105–116, 1998.

[60] G. Freuder and M. Wallace. Constraint technology and the commercial world. *Intelligent Systems and their Applications, IEEE*, 15(1):20–23, 2000.

[61] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15:835–855, 1965.

[62] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[63] M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[64] Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, March 1998.

[65] Fred Glover and Manuel Laguna. General purpose heuristics for integer programming – Part I. *Journal of Heuristics*, 2(4):343–358, 1997.

[66] Fred Glover and Manuel Laguna. General purpose heuristics for integer programming – Part II. *Journal of Heuristics*, 3(2):161–179, 1997.

[67] C. P. Gomes, C. Fernández, B. Selman, and C. Bessière. Statistical Regimes Across Constrainedness Regions. *Constraints*, 10(4):317–337, 2005.

[68] L. Gouveia and P. Pesneau. On extended formulations for the precedence constrained asymmetric traveling salesman problem. *Networks*, 48(2):77–89, 2006.

[69] Andrea Grosso, Marco Locatelli, and Wayne Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *Journal of Heuristics*, 14(6):587–612, 2008.

[70] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2. Springer, 1993.

[71] 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.

[72] Eitan M. Gurari and Ivan Hal Sudborough. Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. *Journal of Algorithms*, 5:531–546, 1984.

[73] G. D. Hachtel and F. Somenzi. A symbolic algorithms for maximum flow in 0-1 networks. *Form. Methods Syst. Des.*, 10(2-3):207–219, April 1997.

[74] T. Hadžić, J. N. Hooker, and P. Tiedemann. Propagating separable equalities in an MDD store. In L. Perron and M. A. Trick, editors, *CPAIOR 2008 Proceedings*, volume 5015 of *Lecture Notes in Computer Science*, pages 318–322. Springer, 2008.

[75] T. Hadzic and J. N. Hooker. Cost-bounded binary decision diagrams for 0-1 programming. In E. Loute and L. Wolsey, editors, *Proceedings of the International Workshop on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2007)*, volume 4510 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2007.

[76] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate Compilation of Constraints into Multivalued Decision Diagrams. In *Proceedings of CP*, volume 5202 of *LNCS*, pages 448–462. Springer, 2008.

[77] T. Hadzic and J.N. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University, 2006.

[78] Tarik Hadzic, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, CP '08, pages 448–462, Berlin, 2008. Springer-Verlag.

[79] W. W. Hager and Y. Krylyuk. Graph partitioning and continuous quadratic programming. *SIAM Journal on Discrete Mathematics*, 12(4):500–523, October 1999.

[80] Utz-Uwe Haus and Carla Michini. Representations of all solutions of boolean programming problems. In *ISAIM*, 2014.

[81] P. Hawkins, V. Lagoon, and P.J. Stuckey. Solving Set Constraint Satisfaction Problems Using ROBDDs. *JAIR*, 24(1):109–156, 2005.

[82] C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization*, 10:673–696, 1997.

[83] István T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. In *Proceedings of Operations Research 2003*, pages 355–362. Springer Verlag, 2003.

[84] S. Hoda, W.-J. van Hoeve, and J. N. Hooker. A Systematic Approach to MDD-Based Constraint Programming. In *Proceedings of CP*, volume 6308 of *LNCS*, pages 266–280. Springer, 2010.

[85] Samid Hoda, Willem-Jan Van Hoeve, and J. N. Hooker. A systematic approach to MDD-based constraint programming. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 266–280, Berlin, Heidelberg, 2010. Springer-Verlag.

[86] John N. Hooker. *Integrated Methods for Optimization (International Series in Operations Research & Management Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 2012.

[87] John N. Hooker. Decision diagrams and dynamic programming. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin Heidelberg, 2013.

[88] 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.

[89] A. J. Hu. Techniques for efficient formal verification using binary decision diagrams. Thesis CS-TR-95-1561, Stanford University, Department of Computer Science, December 1995.

[90] Alexey Ignatiev, Antonio Morgado, Vasco Manquinho, Ines Lynce, and Joao Marques-Silva. Progression in maximum satisfiability. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, IOS Press Frontiers in Artificial Intelligence and Applications, 2014.

[91] ILOG. *CPLEX Optimization Studio V12.4 Manual*, 2012.

[92] 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.

[93] C. Jordan. Sur les assemblages de lignes. *J. Reine Angew Math*, 70:185–190, 1869.

[94] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[95] 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.

[96] Sameer Kumar, Amith R. Mamidala, Daniel Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, Dong Chen, and Burkhard Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *IPDPS-2012: 26th IEEE International Parallel & Distributed Processing Symposium*, pages 763–773, 2012.

[97] Y.-T. Lai, M. Pedram, and S.B.K. Vrudhula. Evbdd-based algorithms for integer linear programming, spectral transformation, and function decomposition. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, pages 959–975, 1994.

[98] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[99] M. Löbbing and I. Wegener. The number of knight's tours equals 13, 267, 364, 410, 532 - counting with binary decision diagrams. *The Eletronic Journal of Combinatorics 3*, #R5.

[100] Tony Lopes, Andre A. Cire, Cid de Souza, and Arnaldo Moura. A hybrid model for a multiproduct pipeline planning and scheduling problem. *Constraints*, 15:151–189, 2010.

[101] M. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-Based Propagators for the SEQUENCE and Related Global Constraints. In *Proceedings of CP*, volume 5202 of *LNCS*, pages 159–174. Springer, 2008.

[102] Rafael Martí, Vicente Campos, and Estefanía Piñana. A branch and bound algorithm for the matrix bandwidth minimization. *European Journal of Operational Research*, 186(2):513–528, 2008.

[103] Rafael Martí, Manuel Laguna, Fred Glover, and Vicente Campos. Reducing the bandwidth of a sparse matrix with tabu search. *European Journal of Operational Research*, 135(2):450–459, 2001.

[104] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th Conference on Design Automation*, pages 272–277. IEEE, 1993.

[105] Shin-ichi Minato. dd: A new decision diagram for efficient problem solving in permutation space. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 90–104. Springer Berlin Heidelberg, 2011.

[106] T. Moisan, J. Gaudreault, and C.-G. Quimper. Parallel Discrepancy-Based Search. In *Proceedings of CP*, pages 30–46, 2013.

[107] N. Narodytska. *Reformulation of Global Constraints*. PhD thesis, University of New South Wales, 2011.

[108] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[109] L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Proceedings of CP*, pages 346–360, 1999.

[110] Estefanía Piñana, Isaac Plana, Vicente Campos, and Rafael Martí. GRASP and path relinking for the matrix bandwidth minimization. *European Journal of Operational Research*, 153(1):200–210, 2004.

[111] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, Third edition, 2008.

[112] Wayne Pullan, Franco Mascia, and Mauro Brunato. Cooperating local search for the maximum clique problem. *Journal of Heuristics*, 17(2):181–199, 2011.

[113] J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of AAAI*, volume 1, pages 362–367. AAAI Press, 1994.

[114] J.-C. Régin. Global Constraints: A Survey. In P. Van Hentenryck and M. Milano, editors, *Hybrid Optimization*, pages 63–134. Springer, 2011.

[115] J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *Proceedings of CP*, volume 1330 of *LNCS*, pages 32–46. Springer, 1997.

[116] J.-C. Régin, M. Rezgui, and A. Malapert. Embarrasingly Parallel Search. In *Proceedings of CP*, volume 8124 of *LNCS*, pages 596–610, 2013.

[117] Andrea Rendl, Matthias Prandtstetter, Gerhard Hiermann, Jakob Puchinger, and Günther Raidl. Hybrid heuristics for multimodal homecare scheduling. In *Proceedings of the 9th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'12, pages 339–355, Berlin, Heidelberg, 2012. Springer-Verlag.

[118] Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations. *Math. Programming*, 121(2):307, 2010.

[119] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[120] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. Report on the experimental language, X10. Technical report, IBM Research, 2011.

[121] J. Saxe. Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM J. Algebraic Discrete Meth.*, 1:363–369, 1980.

[122] Andreas S. Schulz. The permutahedron of series-parallel posets. *Discrete Applied Mathematics*, 57(1):85 – 90, 1995.

[123] Claude E. Shannon. A symbolic analysis of relay and switching circuits. Master's thesis, Massachusetts Institute of Technology, 1937.

[124] Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118(1-4):73–84, 2003.

[125] Michael A. Trick. Sports scheduling. In Pascal van Hentenryck and Michela Milano, editors, *Hybrid Optimization*, volume 45 of *Springer Optimization and Its Applications*, pages 489–508. Springer New York, 2011.

[126] TSPLIB. Retrieved at `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/` on December 10, 2012, 2012.

[127] W.-J. van Hoeve and I. Katriel. Global Constraints. In P. Rossi, F. van Beek and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 6. Elsevier, 2006.

[128] W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. Revisiting the Sequence Constraint. In *Proceedings of CP*, volume 4204 of *LNCS*, pages 620–634. Springer, 2006.

[129] 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.

[130] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of CP-AI-OR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347, Nice, France, April 2004. Springer-Verlag.

[131] Annelie von Arnim, Rainer Schrader, and Yaoguang Wang. The permutahedron of n-sparse posets. *Mathematical Programming*, 75(1):1–18, 1996.

[132] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM monographs on discrete mathematics and applications. Society for Industrial and Applied Mathematics, 2000.

[133] X10 programming language web site. http://x10-lang.org/, January 2010.

[134] Yufei Zhao. The number of independent sets in a regular graph. *Combinatorics, Probability & Computing*, 19(2):315–320, 2010.