Chapter 2
Historical Overview

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

2.1 Introduction

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

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

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

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

2.2 Origins of Decision Diagrams

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

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

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

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

The instruction states: if $x = 0$, go to the instruction at address A, whereas if $x = 1$, go to the instruction at address B. The switching circuit of Fig. 2.1 is represented by the binary-decision program
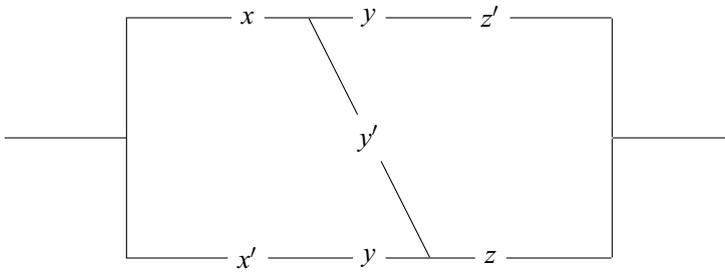
Fig. 2.1 Example of a switching circuit from [110].

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

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

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

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

---

[1] There is such a path, in this case, if one treats the arc from y to 0 as a "long arc," meaning that z can take either value on this arc.
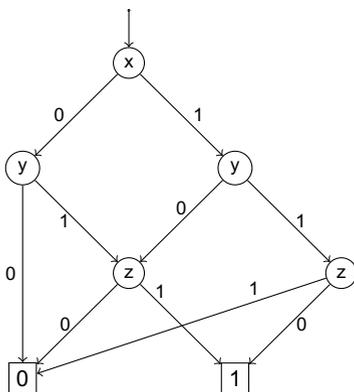
Fig. 2.2 Binary decision diagram corresponding to the binary-decision program (2.1).

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

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

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

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

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

2.3 Decision Diagrams in Optimization

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

2.3.1 Early Applications

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

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

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

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

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

2.3.2 A Discrete Optimization Method

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

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

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

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

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

2.3.3 Decision Diagrams in Constraint Programming

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

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

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

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

2.3.4 Relaxed Decision Diagrams

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

---

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

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

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

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

2.3.5 A General-Purpose Solver

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

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

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

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

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

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

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

2.3.6 Markov Decision Processes

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

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

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

---

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

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

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

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

Springer