Postoptimality Analysis for Integer Programming Using Binary Decision Diagrams

Tarik Hadzic[1] and J. N. Hooker[2]

[1] Cork Constraint Computation Center
t.hadzic@4c.ucc.ie
[2] Carnegie Mellon University
john@hooker.tepper.cmu.edu

Abstract. We show how binary decision diagrams (BDDs) can be used to solve and obtain postoptimality analysis for linear and nonlinear integer programming problems with binary or general integer variables. The constraint set corresponds to a unique reduced BDD that represents all feasible or near-optimal solutions, and in which optimal solutions correspond to certain shortest paths. The BDD can be queried in real time for in-depth postoptimality reasoning. The approach is equally effective for linear and nonlinear problems. There are currently no other methods for obtaining such an analysis, short of repeatedly re-solving the problem. We illustrate the analysis on capital budgeting and network reliability problems.

1 Introduction

Much effort has been invested in the solution of integer programming problems, while postoptimality analysis has received relatively little attention. Yet postoptimality analysis can yield much greater insight into a problem than a single optimal solution. It takes full advantage of the information encoded in an optimization model.

For example, it may be important in a practical setting to characterize the set of optimal or near-optimal solutions, or measure the sensitivity of the optimal value to the problem data. Even more useful would be a tool that responds to what-if queries in real time: what would happen to the optimal value, or the set of near-optimal solutions, if I were to fix certain variables to certain values? Such queries are common in design and configuration problems.

The primary obstacle to postoptimality analysis is its computational intractability. It is difficult to generate or characterize the set of optimal or near-optimal solutions, since this usually requires re-solving the problem many times. Some sensitivity analysis can be obtained from various types of integer programming duality, but these methods yield limited information and may be computationally impractical. What-if queries are not easily supported, again due to the necessity of re-solving the problem repeatedly.

The ideal would be to obtain somehow a compact representation of the set of feasible, optimal, or near-optimal solutions that could be efficiently analyzed and

queried. In this paper we explore the possibility that *binary decision diagrams* (BDDs) can provide such a tool. BDDs (or more properly, *reduced ordered* BDDs) provide a canonical network-based representation of a boolean function (i.e., a two-valued function of two-valued variables). A constraint set in binary variables can be encoded by generating the BDD for the boolean function that is true when the constraints are satisfied and false otherwise. Although the BDD can grow exponentially with the number of variables, in many important cases it provides a surprisingly compact representation of a boolean function. In addition, our recent research [15] shows that if the optimal value (or an approximation of it) is known, the size of the BDD can be substantially reduced by representing only near-optimal solutions, which are the ones of greatest interest.

BDDs have been intensively studied but they have not, to our knowledge, been applied to postoptimality analysis. A key property is that the objective function value of a given solution is the length of a path in the BDD, provided the objective function is separable. Postoptimality analysis can be conducted by analyzing the BDD and its near-shortest paths in various ways.

A particularly attractive feature for integer programming is that BDD-based methods are indifferent to whether the constraints are linear or nonlinear, convex or nonconvex. They need not contain polynomials or any other particular functional form. The objective function can likewise be nonlinear and nonconvex, again provided that it is separable.

BDDs are most readily applied to 0-1 programming because the variables are binary. To accommodate general integer programming, we actually apply the method described here to *multivalued decision diagrams* (MDDs), a straightforward generalization of BDDs. The required MDD can be (a) generated directly, or (b) obtained by encoding the nonbinary variables with binary ones, generating the appropriate BDD, and converting the BDD to an MDD. We take approach (b) due to the availability of BDD software.

In this paper we develop BDD-based algorithms for two types of postoptimality analysis. To illustrate the potential value of the analysis to practitioners, we apply the algorithms to capital budgeting and reliability problems. Both of these problems contain general integer variables, and the reliability problem is highly nonlinear and nonconvex.

One type of postoptimality analysis is *cost-based domain analysis*, which examines the domains of variables in optimal and near-optimal solutions. The domain of a variable is the set of values it can take. The analysis calculates how the domain of a variable 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 the practitioner that there is little choice as to the value of certain variables if one wants a good solution, but there is a good deal of freedom to change the values of other variables.

A related type of analysis is *conditional domain analysis*, which examines domains after restricting a given variable to any value or range of values. One can

also compute sensitivity to right-hand sides, a more traditional type of analysis, if the right-hand side is treated as a variable.

All of these analyses are adaptable to real-time queries, because the necessary information is readily available in the BDD. The user can specify, for example, that certain variables are to be set to certain variables simultaneously, whereupon cost-based and conditional domain analysis is quickly recalculated to reveal the implications of these settings.

Although we use BDDs to solve the problems well as for their postoptimality analysis, we do not take a position here on whether BDDs are appropriate as a general-purpose solution method for integer programming. It may be more effective to solve a problem, or at least estimate its optimal value, by other means. Nonetheless, BDDs should not be ruled out as a solution method for nonlinear problems and situations in which all globally optimal solutions are required.

The paper begins with a brief introduction to BDDs and their application to integer programming problems. It then presents the postoptimality algorithms and applies them to the two problem classes. It concludes with a discussion of computational issues and future research.

2 Previous Work

BDDs have been studied for decades [2, 18]. Bryant [10] showed how to reduce a BDD to canonical form, for a given variable ordering. Readable introductions to BDDs include [3, 11].

Sensitivity analysis for linear integer programming has been examined from several points of view. One is based on integer programming duality, defined in terms of the value function (the optimal value as a function of the right-hand sides) [6–9, 12, 21]. A related method constructs a value function from dual multipliers obtained during a branch-and-bound search [20]. A third, which analyzes sensitivity to constraint coefficients as well as right-hand sides, is based on inference duality and a logical analysis of the constraints [13, 16]. A survey of these methods is presented in [17].

Application of BDDs to optimization has received very limited attention. Becker at al. [5] used BDDs to identify separating cuts for 0-1 linear programming problems in a branch-and-cut context. In [15] we use BDDs to solve integer programming problems by directly representing the space of near-optimal solutions.

3 Binary Decision Diagrams

A BDD is a directed graph that represents a boolean function. A given boolean function corresponds to a unique *reduced BDD* when the variable ordering is fixed. A constraint set in 0-1 variables can be viewed as a boolean function that is true when the constraints are satisfied and false otherwise.

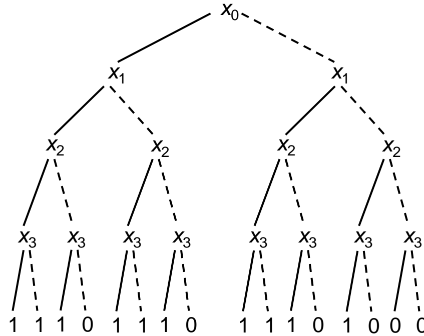

Fig. 1. Branching tree for $2x_0 + 3x_1 + 5x_2 + 5x_3 \geq 7$

The reduced BDD is essentially a compact representation of the branching tree for the constraint set. The leaf nodes of the tree are labeled by 1 or 0 to indicate that the constraint set is satisfied or violated. For example, the tree of Fig. 1 represents the 0-1 linear inequality

$$2x_0 + 3x_1 + 5x_2 + 5x_3 \geq 7 \tag{1}$$

The solid branches (*high edges*) correspond to setting $x_j = 1$ and the dashed branches (*low edges*) to setting $x_j = 0$.

The tree can be transformed to a reduced BDD by repeated application of two operations: (a) if both branches from a node lead to the same subtree, delete the node; (b) if two subtrees are identical, superimpose them. The reduced BDD for (1) appears in Fig. 2(a).

Each path from the root to 1 in a BDD *represents* one or more solutions, namely all solutions in which x_j is set to 0 when the path contains a low edge from a node labeled x_j, and is set to 1 when the path contains a high edge from such a node. A BDD represents the set of all solutions that are represented by a path from the root to 1.

A reduced BDD can in principle be built by constructing the search tree and using intelligent caching to eliminate nodes and superimpose isomorphic subtrees. It is more efficient in practice, however, to combine the BDDs for elementary components of the boolean function. For example, if there are several constraints, one can build a BDD for each constraint and then conjoin the BDDs. Algorithms for building reduced BDDs in this fashion are presented in [3, 15].

The BDD for a linear 0-1 inequality can be surprisingly compact. For instance, the 0-1 inequality

$$\begin{aligned} &300x_0 + 300x_1 + 285x_2 + 285x_3 + 265x_4 + 265x_5 + 230x_6 + \\ &\quad 23x_7 + 190x_8 + 200x_9 + 400x_{10} + 200x_{11} + 400x_{12} + \\ &\quad 200x_{13} + 400x_{14} + 200x_{15} + 400x_{16} + 200x_{17} + 400x_{18} \geq 2701 \end{aligned} \tag{2}$$

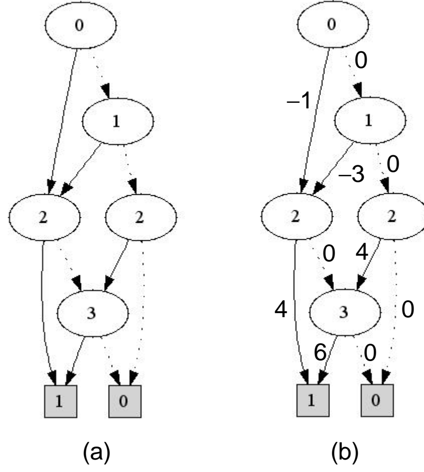

Fig. 2. (a) Reduced BDD for $2x_0 + 3x_1 + 5x_2 + 5x_3 \geq 7$ using the variable ordering x_0, x_1, x_2, x_3. (b) Same BDD with edge lengths corresponding to the objective function $2x_0 - 3x_1 + 4x_2 + 6x_3$

has a complex feasible set that contains 117,520 minimally feasible solutions (each of which becomes infeasible if any variable is flipped from 1 to 0), as reported in [4]. (Equivalently, if the right-hand side is ≤ 2700, the inequality has 117,520 minimal covers.) The BDD for (2) contains only 152 nodes.

A separable objective function $\sum_j c_j(x_j)$ can be minimized subject to a constraint set by finding a shortest path from the root to 1 in the corresponding BDD. Let $var[u] = i$ when variable x_i labels node u. Then if $var[u] = k$ and $var[u'] = \ell$, a high edge from u to u' has length $c[u, u'] = c_k(1) + c^*_{k+1,\ell-1}$, where $c^*_{k+1,\ell-1}$ is the cost of setting every skipped variable $x_{k+1}, \ldots, x_{l-1}$ to a value that gives the lowest cost. To make this more precise, let $v_{uu'}$ be 1 if (u, u') is a high edge and 0 if it is a low edge. Then

$$c[u, u'] = c_k(v_{uu'}) + c^*_{k+1,\ell-1}$$

and

$$c^*_{pq} = \sum_{j=p}^{q} \min\{c_j(1), c_j(0)\}$$

For example, if we minimize

$$2x_0 - 3x_1 + 4x_2 + 6x_3 \tag{3}$$

subject to (1), the associated BDD has the edge lengths shown in Fig. 2(b). Note that the length of the high edge from the root node is $c_0(1) + c^*_{11} = 2 - 3 = -1$. The shortest path from the root node to 1 has length 1 and passes through the x_1 node and the x_2 node on the left. Its three edges indicate that $(x_0, x_1, x_2) =$

$(0, 1, 1)$. This corresponds to optimal solution $(x_0, x_1, x_2, x_3) = (0, 1, 1, 0)$, where x_3 is set to zero to minimize the x_3 term in the objective function.

4 Multivalued Decision Diagrams

If some variables can take more than two values, as in general integer programming, a *multivalued decision diagram* (MDD) represents the solution space in a manner that is precisely analogous to BDD representation. An MDD differs from a BDD only in that a node can have more than two outgoing edges, representing possible values of the corresponding variable. Thus if node u is labeled by variable x_i, each edge (u, u') corresponds to a value $v_{uu'} \in D_i$, where D_i is the domain of x_i. An optimal solution is again found by computing a shortest path to 1 in the MDD. Edge costs $c[u, u']$ are computed as before except that

$$c^*_{pq} = \sum_{j=p}^{q} \min_{v' \in D_j} \{c_j(v')\}$$

An MDD can in principle be constructed directly from the problem instance, but we found it convenient to build a BDD first, using available software, and then convert the BDD to an MDD. The BDD, which we call B, is constructed by first encoding each variable having k possible values as a tuple of $\lceil \log_2 k \rceil$ binary variables and then building B for the binary variables in the usual fashion.

Conversion of B to an MDD proceeds as follows. Let L_i be the set of the nodes in B that are labeled with one of the variables used to encode x_i. Let In_i be the set of nodes in L_i directly reachable from outside L_i. That is, In_i is the set of nodes $u' \in L_i$ such that $[u, u']$ is an edge of B for some $u \notin L_i$. We initialize the process by letting In_1 consist of the root node. The nodes in In_i become the nodes on level i of the MDD (i.e., the nodes u with $var[u] = i$). The MDD contains an edge $[u, u']$ if and only if $u \in In_i$ for some i and there is a path P from u to u' in B that encodes a binary representation of x_i. That is, all the nodes on P except u' are labeled with variables that encode x_i.

5 Projection and Postoptimality Analysis

Consider a 0-1 programming problem with a separable objective function:

$$\begin{aligned} &\min \sum_{j=1}^{n} c_j(x_j) \\ &g_i(x) \geq b_i, \ \ i = 1, \ldots, m \\ &x_j \in \{0, 1\}, \ \ j = 1, \ldots, n \end{aligned} \tag{4}$$

We refer to any 0-1 n-tuple as a *solution* of (4), any solution that satisfies the constraints as a *feasible solution*. If Sol is the set of feasible solutions, and c^* is

the optimal value, then for a given tolerance Δ the set of near-optimal feasible solutions of (4) is

$$Sol_\Delta = \left\{ x \in Sol \;\middle|\; \sum c_j(x_j) \le c^* + \Delta \right\}$$

In general, *cost-based domain analysis* derives the projection $Sol_{\Delta i}$ of Sol_Δ onto any x_i for any Δ between 0 and some maximum tolerance $\Delta_{\max}$. It may also incorporate conditional domain analysis subject to a partial assignment $x_J = v_J$ specified by the user, where $J \subset \{1, \ldots, n\}$. This projection is

$$Sol_{\Delta i}(x_J = v_J) = \{x_i \mid x \in Sol_\Delta, x_J = v_J\}$$

Conditional domain analysis computes the projection of the feasible set onto each x_i, given that $x_J = v_J$. Thus it computes

$$Sol_i(x_J = v_J) = \{x_i \mid x \in Sol, x_J = v_J\}$$

6 MDD-Based Postoptimality Analysis

Cost-based domain analysis requires computing $Sol_{\Delta i}$. This can be accomplished by computing, for each $v \in D_i$, the *minimal feasible cost* $c^*(i, v)$ given that $x_i = v$. This is the minimum value of (4) with the additional constraint that $x_i = v$. Then $v \in Sol_{\Delta i}$ if and only if $c^*(v, i) \le c^* + \Delta$.

To compute $c^*(i, v)$, let $\mathrm{SP}[u, u']$ be the length of a shortest path in the MDD from the root to 1 through $[u, u']$. Then

$$\mathrm{SP}[u, u'] = U[u] + c[u, u'] + D[u']$$

where $U[u]$ and $D[u']$ denote the length of a shortest path from u up to the root, and from u' down to 1, respectively. Now the length of a shortest path on which x_i is fixed to v is

$$\mathrm{SP}(i, v) = \min_{(u,u')} \{\mathrm{SP}[u, u'] \mid var[u] = i\}$$

Also the length of a shortest path that skips all x_i nodes is

$$\mathrm{SP}(i) = \min_{(u,u')} \{\mathrm{SP}[u, u'] \mid var[u] < i < var[u']\}$$

Finally,

$$c^*(i, v) = \min \left\{ \mathrm{SP}(i, v),\ \mathrm{SP}(i) + c_i(v) - \min_{v' \in D_i} \{c_i(v')\} \right\} \tag{5}$$

An algorithm appears in Fig. 3. The subroutine *shortestPaths*$[u]$ calculates the length of a shortest path from each node to u.

Conditional postoptimality analysis requires computing $Sol_i(x_J = v_J)$. But $v_i \in Sol_i(x_J = v_J)$ if and only if there is a path from the root to 1 such that for each $k \in J \cup \{i\}$, (a) some edge (u, u') with $\mathrm{var}[u] = k$ and $v_{uu'} = v_k$ lies on the path, or (b) the path contains no node u with $\mathrm{var}[u] = k$.

```
1: for i = 1,...,n, SP(i) ← ∞
2:   for all v ∈ D_i, SP(i,v) ← ∞
3: U ← shortestPaths[root]
4: D ← shortestPaths[1]
5: for all edges (u,u')
6:   SP[u,u'] ← min{SP[u,u'], c[u,u']}
7:   for i = var[u] + 1 to var[u'] − 1
8:      SP(i) ← min{SP(i), c[u,u']}
```

Fig. 3. Algorithm for computing the minimum feasible cost $c^*(i,v)$ using (5). The worst-case execution time is $O(mn + \sum_{i=1}^{n} |D_i|)$, where n is the number of variables and m the number of edges in the MDD.

7 Example: Capital Budgeting

We now apply cost-based and conditional domain analysis to two problem classes. The aim is to illustrate the potential value of such analysis to practitioners. We begin with a simple capital budgeting problem.

We are given a set of n investment opportunities, perhaps types of manufacturing plants. A plant of type i costs a_i and yields a present value return of c_i. We can potentially build several plants of a given type. Given the total budget b, the capital budgeting problem asks how we can maximize return subject to the budget limitation:

$$\begin{aligned}
&\max \sum_{i=1}^{n} c_i x_i \\
&\sum_{i=1}^{n} a_i x_i \leq b \\
&x_i \in D_i, \ i = 1, \ldots, n
\end{aligned}$$

Since the objective is to maximize, we seek a longest path rather than a shortest path in the MDD.

Consider an example with $n = 10$ variables in which

$$\begin{aligned}
&c = (503, 493, 450, 410, 395, 389, 360, 331, 304, 299) \\
&a = (249, 241, 219, 211, 194, 196, 177, 162, 150, 149), \quad b = 1800 \\
&D_i = \{0, \ldots, 3\}, \text{ all } i
\end{aligned}$$

This instance compiles into a BDD of 1080 nodes. The corresponding MDD has 542 vertices and 1933 edges. A cost-based domain analysis appears in Table 1. In a column corresponding to variable x_i, the table displays values that are *added* to $Sol_{\Delta i}$ as Δ increases. Thus x_1 must take the value 0 if the maximum return of 3678 is to be achieved. If we permit a return that is within $\Delta = 5$ of the maximum, then x_1 can take any of the values in $Sol_{\Delta 1} = \{0, 1, 2\}$. Similarly, x_3 can take any of the values in $Sol_{\Delta 3} = \{1, 2, 3\}$.

Note that there may not be a solution within $\Delta = 5$ of the maximum in which $x_1 = 2$ *and* $x_3 = 3$. Since the table displays projections onto each variable, it

Table 1. Cost-based domain analysis for the capital budgeting example.

Δ	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0	0	3	2	0	0	0	1	1	2	0
1			1		1			3	0	
2							0		1	
5	1,2	0,1,2	3		3		2,3	2	3	1
6					2			0		
9										2
10			0							
12						1				
14										3
20				1						
21	3					2				
32						3				
45				2						
62				3						

tells us only that there is a solution within 5 of the maximum in which $x_1 = 2$, and one in which $x_3 = 3$. It is straightforward to query the MDD, however, to determine whether there is a solution in which $(x_1, x_3) = (2, 3)$ by computing $Sol_{\Delta 1}(x_3 = 3)$.

Even in this simple example we can deduce some useful facts. There is exactly one optimal solution, since each $Sol_{\Delta i}$ is a singleton for $\Delta = 0$. Relaxing the tolerance to $\Delta = 5$ results in a great deal of freedom in choosing investment strategies. Investments 4 and 6 are highly undesirable, since building one or more of them requires substantial revenue loss. Yet the return from building copies of a given plant need not be monotonic. Building two copies of plant 3, for example, is better than building one copy, which is better than building three copies.

8 Network Reliability

The reliability of a complex system composed of n subsystems can be improved by increasing the redundancy of each subsystem; that is, by increasing the number of components that implement the same functionality. Suppose that each component of a subsystem i has reliability $r_i \in [0, 1]$. If x_i denotes the number of parallel components, the reliability of the subsystem is

$$R_i(x_i) = 1 - (1 - r_i)^{x_i}$$

The reliability of the entire system can be expressed as an algebraic expression $R(R_1, \ldots, R_n)$ over reliabilities of each of its subsystems. The methods for deriving such expressions are well described [1, 14].

Each additional component of subsystem i incurs a unit cost c_i. We suppose that the system can be represented as a network in which each edge e_i represents

a subsystem i that consists of x_i redundant components. The reliability of the entire system is the probability that at least one component is working on each edge along some path from vertex s (source) to vertex t (terminal).

The network reliability problem can be formulated

$$
\begin{aligned}
&\min \sum_{i=1}^{n} c_i x_i \\
&R(R_1(x_1), \ldots, R_n(x_n)) \geq Rel \\
&Rel \in [0, 1] \\
&x_i \in D_i,\ i = 1, \ldots, n
\end{aligned}
$$

where the minimal acceptable reliability Rel is treated as a variable. Coefficients in the reliability constraint are converted to integers by multiplying the constraint by 100 or 1000 and truncating any roundoff error. The domain of Rel is similarly converted to integers. Thus $Rel \in \{0, \ldots, 100\}$ or $Rel \in \{0, \ldots, 1000\}$, although in the tables to follow, Rel is given as a percentage. Thus if we scale with a factor of 1000 and $Rel = 955$, the minimal reliability level is given as 99.5%.

We analyze network reliability instances from the literature [1, 19] that have five, seven, and twelve edges (Rel5, Rel7, and Rel12, respectively). The networks appear in Figs. 4–5, where the designated vertices s and t are always the leftmost and rightmost vertices, respectively. The compilation statistics are reported in Table 2. Computation times are as observed on a Pentium-III 1 GHz machine with 2GB of memory, running Linux.

Figure 4 shows the Rel5 network. The component reliabilities are $r = (0.9, 0.85, 0.8, 0.9, 0.95)$, and the corresponding costs are $c = (25, 35, 40, 10, 60)$. The reliability expression for the entire system is

$$
\begin{aligned}
R(R_1, \ldots, R_5) = {} & R_1R_2 + (1 - R_2)R_3R_4 + (1 - R_1)R_2R_3R_4 \\
& + R_1(1 - R_2)(1 - R_3)R_4R_5 + (1 - R_1)R_2R_3(1 - R_4)R_5
\end{aligned}
$$

The expression is highly nonconvex due to the products of variables. The complexity of the expression grows rapidly for larger networks.

Table 3(a) shows cost-based domain analysis with the minimal reliability level Rel set at 60%. The unique minimum cost solution uses only edges 3 and 4. It actually achieves reliability of 72%, at a cost of 50. Higher reliabilities cost more, up to 170 for 99% reliability. The main value of this analysis is to show the tradeoff between cost and reliability.

Table 2. Compilation statistics for network reliability instances.

	BDD		*MDD*		*Time*
Instance	*Nodes*	*Edges*	*Nodes*	*Edges*	*(sec)*
Rel5	308	616	89	766	1.2
Rel7	7,779	15,558	3,126	89,770	9.0
Rel12	69,457	138,914	36,301	139,744	1980

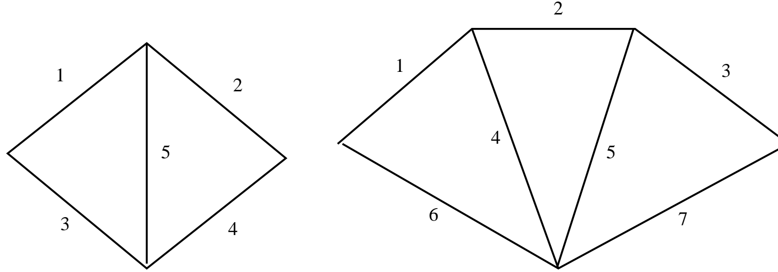

Fig. 4. Reliability instances Rel5 (left) and Rel7 (right).

Table 3. Cost-based domain analysis (a) and conditional domain analysis (b) for Rel5, with $c^* = 50$.

(a)

$c^* + \Delta$	x_1	x_2	x_3	x_4	x_5	*Rel*
50	0	0	1	1	0	72
60	1	1	0	0,2		79
85	2					84
90			2	3		86
95		2			1	88
100						95
120						97
125	3					
155		3			2	
160						98
170						99
180			3			
230					3	

(b)

Rel	x_1	x_2	x_3	x_4	x_5
99	1,2	1,2	1,2	1,2	0,1,2,3
98		0,3	0,3		
97				0,3	
95	0,3				

A conditional domain analysis for variable *Rel* appears in Table 3(b). Note that there is a solution that achieves 99% reliability without using edge 5. Edge x_2 need not appear in a solution with 98% reliability, and similarly for edges 3 and 4. Edge 1 becomes dispensable only when reliability is reduced to 95%.

The reliabilities for Rel 7 (Figure 4) are $r = (0.7, 0.9, 0.8, 0.65, 0.7, 0.85, 0.85)$ and the costs are $c = (4, 5, 4, 3, 3, 4, 5)$. The minimal reliability level *Rel* is set to 80%. Cost-based and conditional domain analyses appear in Table 4. Edges 6 and 7 are more important for high reliability, which reflects the fact that they form the shortest s-t path. Edge 6 is most critical, since its omission reduces reliability to 97.2%. Edge 2 is the least critical because 99.9% reliability can be achieved without it, even though its components have the highest reliability level of 90%.

Network Rel12 appears in Figure 5. The reliabilities are r = (0.95, 0.90, 0.70, 0.75, 0.85, 0.85, 0.85, 0.85, 0.90, 0.95, 0.90, 0.80) and the costs are $c = (50, 60, 25, 20, 45, 50, 30, 10, 45, 20, 50, 80)$. Minimum reliability is again 80%. Cost-based domain analysis appears in Table 5(a). There is a unique minimum-cost path 3-1-4-8, and none of these edges can be deleted without substantial increase in cost if 80% reliability is to be maintained. Edge 3 or 1, for example, cannot

Table 4. Cost-based domain analysis (a) and conditional domain analysis (b) for Rel7, with $c^* = 9$.

(a)

$c^* + \Delta$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	*Rel*
9	0	0	0	0	0	1	1	72.2
11			1		1		0	
12	1			1		0		
13		1				2		82.9
14					2		2	
15			2	2				
16	2							
17					3	3		84.6
18		2		3				95.2
19			3				3	
20	3							
22								97.2
23		3						
27								99.2
34								99.4
40								99.6
43								99.7
47								99.8
54								99.9

(b)

Rel	x_1	x_2	x_3	x_4	x_5	x_6	x_7
99.9	2,3	0..3	1,2,3	0..3	0..3	2,3	2,3
99.8	1						1
99.5	0		0			1	
99.1							0
97.2						0	

be deleted without raising the cost from 180 to 230. A good deal more design freedom is possible when cost is 230 or greater.

A conditional domain analysis appears in Table 5(b). Note that edges 3, 11, 8 and 12 are most critical for high reliability. This is intuitively plausible, because they are the edges that are incident to s and t.

Although the MDD for Rel12 is modest in size, the compilation time (over 30 minutes) is much greater than for Rel5 and Rel7 (a few seconds). Yet once the MDD is constructed, the time required to query it is very small. The total query time for constructing Table 5 was only 1.8 seconds. Also, one can reduce the compilation time substantially by representing only near-optimal solutions, as discussed in the next section.

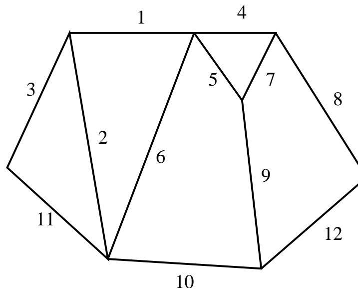

Fig. 5. Reliability instance Rel12.

Table 5. Cost-based domain analysis (a) and conditional domain analysis (b) for Rel12, with $c^* = 180$.

(a)

$c^* + \Delta$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	*Rel*
180	1	0	2	3	0	0	0	2	0	0	0	0	80
185			3	2									82
190								3					
195													83
200										1			
205													86
210	2						1						
215													88
220										2			
225					1				1				
230	0		0			1,2					1		
235			1										
240		1		1			2			3	2	1	
250				0				0,1				2	
255													91
260	3								2				
265													93
270					2	3	3						
290											3		
300		2											
305									3				
310												3	
315					3								94
340													95
360		3											
365													96
380													97
430													98
485													99

(b)

Rel	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
99	0..3	0..3	1..3	0..3	0..3	0..3	0..3	1..3	0..3	0..3	1..3	1..3
98			0					0				0
96											0	

9 Computational Issues and Future Work

We have shown how binary decision diagrams can provide real-time and in-depth postoptimality analysis for linear and nonlinear integer programming problems. To our knowledge, no other technique provides postoptimality analysis at this level of detail, aside from repeatedly re-solving the problem.

Once the BDD is generated, it can be queried for postoptimality analysis in negligible time. An important research issue is whether the size of the BDD, and the computational cost of generating it, remain reasonable as the problem scales up. For the purposes of this paper we used BDDs that represent the entire feasible set. The BDD sizes were well within practical size limits for the problems problems studied here, but a full BDD can explode for larger problems. The compilation time can also grow rapidly even if the BDD is rather small, as we saw with problem Rel12.

Fortunately, for most purposes it suffices to use a *cost bounded* BDD that represents only near-optimal solutions. This is, the BDD encodes all feasible solutions with value less than or equal to an upper bound $\bar{c}$, which is set to be within some tolerance $\Delta_{\max}$ of what is known or estimated to be the optimal value c^*. Although the cost bounded BDD can be as large as, or even larger than, the original BDD, we show in [15] how to generate quickly a much smaller *sound* BDD that approximates the cost bounded BDD and is valid for postoptimality analysis. Computational testing on linear 0-1 problems having up to 60 variables and 10 constraints shows that the sound BDD is much smaller than the original BDD for rather large tolerances $\Delta_{\max}$. It is quite small even when the full BDD is too large to compute. BDDs also easily found all optimal solutions for a sampling of MIPLIB problems (lseu, p0033, p0201, stein27, stein45) when the optimal values were given.

These results suggest that BDD-based postoptimality analysis can be computationally practical at least for medium-sized integer programming problems, provided the optimal value c^* can be estimated, or perhaps obtained by computing an initial optimal solution with another method. The BDD approach can solve the problem from scratch if the estimated value $\bar{c}$ is at least as great as c^* and is close enough to c^* to result in a sound BDD of reasonable size.

Future research should include computational testing of BDDs for a variety of problem classes, for purposes of both solution and postoptimality analysis. A particularly interesting issue is how information-rich BDDs can be better exploited to yield new types of postoptimality analysis.

References

1. J. A. Abraham. An improved algorithm for network reliability. *IEEE Transactions on Reliability*, 28(1):58–61, 1979.
2. S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.
3. H. R. Andersen. An introduction to binary decision diagrams. Lecture notes, available online, IT University of Copenhagen, 1997.
4. P. Barth. *Logic-based 0-1 Constraint Solving in Constraint Logic Programming.* Kluwer, Dordrecht, 1995.
5. 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.
6. C. E. Blair and R. G. Jeroslow. The value function of a mixed integer program: I. *Discrete Applied Mathematics*, 19:121–138, 1977.
7. C. E. Blair and R. G. Jeroslow. The value function of a mixed integer program. *Mathematical Programming*, 23:237–273, 1982.
8. C. E. Blair and R. G. Jeroslow. Constructive characterizations of the value function of a mixed-integer program I. *Discrete Applied Mathematics*, 9:217–233, 1984.
9. C. E. Blair and R. G. Jeroslow. Constructive characterizations of the value function of a mixed-integer program II. *Discrete Applied Mathematics*, 10:227–240, 1985.

10. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
11. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293– 318, 1992.
12. W. Cook, A. M. H. Gerards, A. Schrijver, and E. Tardos. Sensitivity theorems in integer linear programming. *Mathematical Programming*, 34:251–264, 1986.
13. M. Dawande and J. N. Hooker. Inference-based sensitivity analysis for mixed integer/linear programming. *Operations Research*, 48:623–634, 2000.
14. Luigi Fratta and Ugo G. Montanari. Boolean algebra method for computing the terminal. reliability in a communication. network. *IEEE Transactions on Circuit Theory*, 20(3):203–211, 1973.
15. 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.
16. J. N. Hooker. Inference duality as a basis for sensitivity analysis. In E. C. Freuder, editor, *Principles and Practice of Constraint Programming (CP 1996)*, volume 1118 of *Lecture Notes in Computer Science*, pages 224–236. Springer, 1996.
17. J. N. Hooker. Integer programming duality. In C. A. Floudas and P. M. Pardalos, editors, *Encyclopedia of Optimization, Vol. 2*, pages 533–543. Kluwer, New York, 2001.
18. C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.
19. K. B. Misra and U. Sharmma. An efficient algorithm to solve integer-programming problems arising in system-reliability design. *IEEE Transactions on Reliability*, 40(1):81–91, 1991.
20. L. Schrage and L. Wolsey. Sensitivity analysis for branch and bound integer programming. *Operations Research*, 33:1008–1023, 1985.
21. L. A. Wolsey. Integer programming duality: Price functions and sensitivity analysis. *Mathematical Programming*, 20:173–195, 1981.