Optimization Bounds from Binary Decision Diagrams

David Bergman Andre A. Cire Willem-Jan van Hoeve
J. N. Hooker
Tepper School of Business, Carnegie Mellon University
dbergman,acire,vanhoeve,jh38andrew.cmu.edu

Revised February 2013

Abstract

We explore the idea of obtaining bounds on the value of an optimization problem from a discrete relaxation based on binary decision diagrams (BDDs). We show how to construct a BDD that represents a relaxation of a 0–1 optimization problem, and how to obtain a bound for a separable objective function by solving a shortest (or longest) path problem in the BDD. As a test case we apply the method to the maximum independent set problem on a graph. We find that for most problem instances, it delivers tighter bounds, in less computation time, than state-of-the-art integer programming software obtains by solving a continuous relaxation augmented with cutting planes.

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 paper, we explore an alternative strategy of obtaining bounds from a *discrete* relaxation, namely a binary decision diagram (BDD).

Binary decision diagrams are compact graphical representations of Boolean functions (Akers 1978, Lee 1959, Bryant 1986). They were originally introduced for applications in circuit design and formal verification (Hu 1995, Lee 1959) but have since been used for a variety of other purposes. These include sequential pattern mining and genetic programming (Loekito et al. 2010, Wegener 2000).

A BDD can represent the feasible set of a 0-1 optimization problem, because the constraints can be viewed as defining a Boolean function $f(x)$ that is 1 when x is a feasible solution. Unfortunately, a BDD that exactly represents the feasible set can grow exponentially in size. We circumvent this difficulty by creating a *relaxed* BDD of limited size that represents a superset of the feasible set. The relaxation is created by merging nodes of the BDD in such a way that no feasible solutions are excluded. A bound on any additively separable objective function can now be obtained by solving a longest (or shortest) path problem on the relaxed BDD. The idea is readily extended to general discrete (as opposed to 0-1) optimization problems by using *multivalued decision diagrams* (MDDs).

As a test case, we apply the proposed method to the maximum independent set problem on a graph. We find that BDDs can deliver tighter bounds than those obtained by a strong LP formulation, even when the LP is augmented by cutting planes generated at the root node by a state-of-the-art mixed integer solver. In most instances, the BDD bounds are obtained in less computation time, even though we used a non-default barrier LP solver that is faster for these instances.

The paper is organized as follows. After a brief literature review, we show how BDDs can represent 0-1 optimization problems in general and the maximum weighted independent set problem in particular. We then exhibit an efficient top-down compilation algorithm that generates exact reduced BDDs for the independent set problem, and prove its correctness. We then modify the algorithm to generate a limited-size relaxed BDD, prove its correctness, and show that it has polynomial time complexity. We also discuss variable ordering for exact and relaxed BDD compilation, as this can have a significant impact on the size of the exact BDD and the bound provided by relaxed BDDs. In addition, we describe heuristics for deciding which nodes to merge while building a relaxed BDD.

At this point we report computational results for random and benchmark instances of the maximum independent set problem. We experiment with various heuristics for ordering variables and merging nodes in the relaxed

BDDs and test the quality of the bound provided by the relaxed BDDs versus the maximum BDD size. We then compare the bounds obtained from the BDDs with the LP bounds obtained by a commercial mixed integer solver. We conclude with suggestions for future work.

2. Previous Work

Relaxed BDDs and MDDs were introduced by Andersen et al. (2007) for the purpose of replacing the domain store used in constraint programming by a richer data structure. They found that MDDs drastically reduce the size of the search tree and allow faster solution of problems with multiple all-different constraints, which are equivalent to graph coloring problems. Similar methods were applied to other types of constraints in Hadzic et al. (2008) and Hoda et al. (2010). The latter paper also develops a general top-down compilation method based on state information at nodes of the MDD.

None of this work addresses the issue of obtaining bounds from relaxed BDDs. Three of us applied this idea to the set covering problem in a conference paper (Bergman et al. 2011), which reports good results for certain structured instances. In the current paper, we present novel and improved methods for BDD compilation and relaxation. These methods are superior to continuous relaxation technology for a much wider range of instances, and require far less time.

The ordering of variables can have a significant bearing on the effectiveness of a BDD relaxation. We investigated this for the independent set problem in Bergman et al. (2012) and apply the results here.

We note that binary decision diagrams have also been applied to post-optimality analysis in discrete optimization (Hadzic and Hooker 2006, 2007), cut generation in integer programming (Becker et al. 2005), and 0-1 vertex and facet enumeration (Behle and Eisenbrand 2007).

Branch-and-bound methods for the independent set problem, which make essential use of relaxation bounds, are studied by Rossi and Smriglio (2001), Tomita and Kameda (2007), Rebennack et al. (2011), and surveyed by Rebennack et al. (2012).

3. Binary Decision Diagrams

Given binary variables $x = (x_1, \ldots, x_n)$, a *binary decision diagram* (BDD) $B = (U, A)$ for x is a directed acyclic multigraph that encodes a set of values of x. The set U of nodes is partitioned into layers $L_1, \ldots, L_n$ corresponding to variables $x_1, \ldots, x_n$, plus a terminal layer L_{n+1}. Layers L_1 and L_{n+1} are singletons consisting of the *root* node r and the *terminal* node t, respectively. All directed arcs in A run from a node in some layer L_j to a node in some deeper layer L_k ($j < k$). For a node $u \in L_j$, we write $\ell(u) = j$ to indicate the layer in which u lies.

Each node $u \in L_j$ has one or two out-directed arcs, a *0-arc* $a_0(u)$ and/or a *1-arc* $a_1(u)$. These correspond to setting x_j to 0 and 1, respectively. We use the notation $b_0(u)$ to indicate the node at the opposite end of arc $a_0(u)$, and similarly for $b_1(u)$. Thus, 0-arc $a_0(u)$ is $(u, b_0(u))$, and 1-arc $a_1(u)$ is $(u, b_1(u))$. Each arc-specified path from r to t represents the 0-1 tuple x in which $x_{\ell(u)} = 1$ for each 1-arc $a_1(u)$ on the path, and $x_j = 0$ for all other j. The entire BDD represents the set $\mathrm{Sol}(B)$ of all tuples corresponding to r–t paths.

It is often useful to abbreviate a BDD by using *long arcs*. These arcs skip over variables whose values are represented implicitly. A long arc can indicate that all skipped variables take the value zero (resulting in a *zero-suppressed* BDD) or the value one (a *one-suppressed* BDD). More commonly, a long arc indicates that the skipped variables can take either value. One advantage of BDDs is that we can choose the type of long arc that suits the problem at hand. We use zero-suppressed BDDs (Minato 1993) because there are many zero-valued arcs in BDDs for the independent set problem. Thus a long arc from layer L_j to layer L_k encodes the partial assignment $(x_j, \ldots, x_{k-1}) = (1, 0, \ldots, 0)$.

Figure 1(b) illustrates a BDD for variables $x = (x_1, \ldots, x_6)$. The left-most path from root node r to terminal node t represents the tuple $(x_1, \ldots, x_6) = (0, 0, 1, 0, 0, 0)$. The third arc in the path is a long arc because it skips three variables. It encodes the partial assignment $(x_3, x_4, x_5, x_6) = (1, 0, 0, 0)$. The entire BDD of Fig. 1(b) represents a set of 10 tuples, corresponding to the 10 r–t paths.

Given nodes $u, u' \in U$, we will say that $B_{uu'}$ is the portion of B induced by the nodes in U that lie on some directed path from u to u'. Thus $B_{rt} = B$. Two nodes u, u' on a given layer of a BDD are *equivalent* if B_{ut} and $B_{u't}$ are the same BDD. A *reduced* BDD is one that contains no equivalent nodes. A

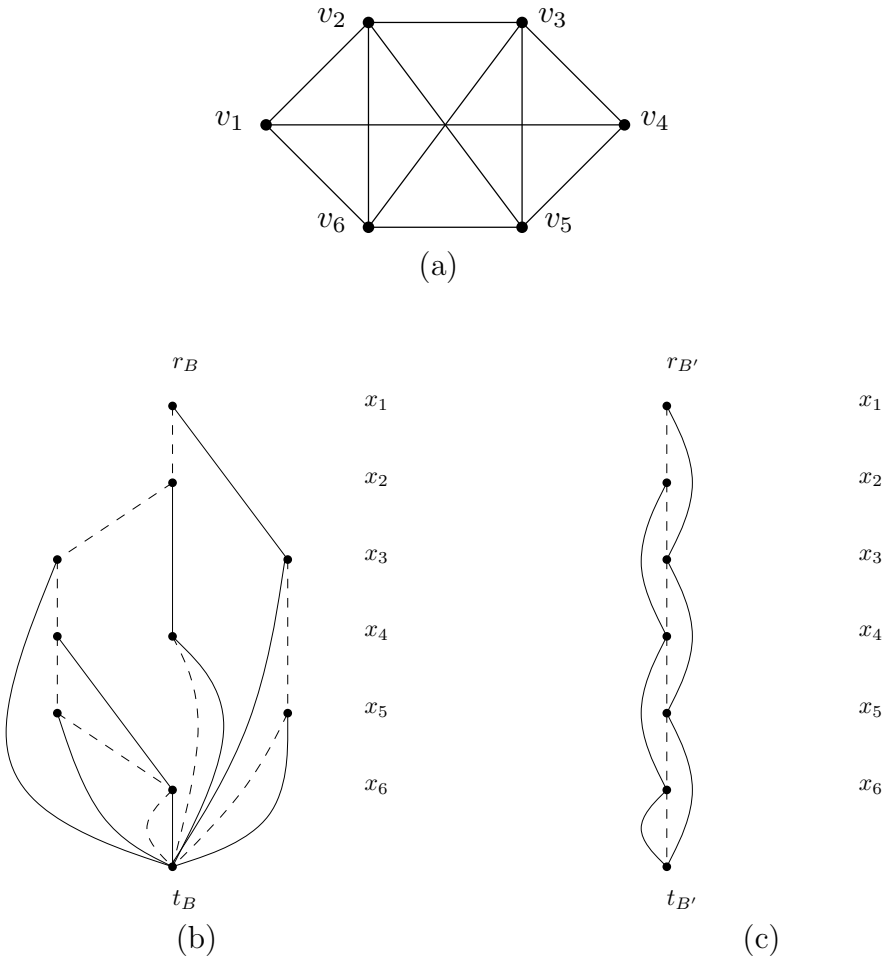


Figure 1: (a) Instance of the independent set problem. (b) Exact BDD for the instance. (c) Relaxed BDD for the instance.

standard result of BDD theory Bryant (1986), Wegener (2000) is that for a fixed variable order, there is a unique reduced BDD that represents a given set. The *width* ω_j of layer L_j is $|L_j|$, and the width $\omega(B)$ of a BDD B is $\max_j\{\omega_j\}$. The BDD of Fig. 1(a) is reduced and has width 2.

The feasible set of any optimization problem with binary variables $x_1, \ldots, x_n$ can be represented by an appropriate reduced BDD. The BDD can be regarded as a compact representation of a search tree for the problem. It can in principle be obtained by omitting infeasible leaf nodes from the search tree, superimposing isomorphic subtrees, and identifying all feasible leaf nodes

with t. We will present below a much more efficient procedure for obtaining a reduced BDD. A slight generalization of BDDs, *multivalued decision diagrams* (MDDs), can similarly represent the feasible set of any discrete optimization problem. MDDs allow a node to have more than two outgoing arcs and therefore accommodate discrete variables with several possible values.

4. BDD Representation of Independent Sets

We focus on BDD representations of the *maximum weighted independent set problem*. Given a graph $G = (V, E)$, an *independent set* is a subset of the vertex set V, such that no two vertices are connected by an edge in E. If each vertex v_j is associated with a weight w_j, the problem is to find an independent set of maximum weight. If each $w_j = 1$, we have the *maximum independent set problem*.

If we let binary variable x_j be 1 when v_j is included in the independent set, the feasible solutions of any instance of the independent set problem can be represented by a BDD on variables $x_1, \ldots, x_n$. Figure 1(b), for example, represents the 10 independent sets of the graph in Fig. 1(a).

We can remove any node u in a BDD with a single outgoing arc if it is a 0-arc $a_0(u)$. This is accomplished by replacing every 0-arc $a_0(u')$ for which $b_0(u') = u$ with a longer arc $a_0(u')$ for which $b_0(u') = b_0(u)$, and similarly for every such 1-arc. If the BDD represents an instance of the independent set problem, a single outgoing arc must be a 0-arc, which means that all nodes with single outgoing arcs can be removed. Every node in the resulting BDD has exactly two outgoing arcs.

To represent the objective function in the BDD, let each 1-arc $a_1(u)$ have length equal to the weight $w_{\ell(u)}$, and each 0-arc length 0. Then the length of a path from r to t is the weight of the independent set it represents. The weighted independent set problem becomes the problem of finding a longest path in a BDD. If for all vertices v_j weight $w_j = 1$, the four longest paths in the BDD of Fig. 1(b) have length 2, corresponding to the maximum independent sets $\{v_1, v_3\}$, $\{v_1, v_5\}$, $\{v_2, v_4\}$, and $\{v_4, v_6\}$.

Any binary optimization problem with an additively separable objective function $\sum_j f_j(x_j)$ can be similarly represented as a longest path problem on a BDD. Zero-suppressing long edges may be used if $f_j(0) = 0$ and $f_j(1) \geq 0$ for each j. This condition is met by any independent set problem with nonnegative weights. It can be met by any binary problem if each $f_j(x_j)$ is

replaced with $\bar{f}(\bar{x}_j)$, where $\bar{f}_j(0) = 0$ and

$$\begin{array}{ll} \bar{f}_j(1) = f_j(1) - f_j(0) \text{ and } \bar{x}_j = x_j, & \text{if } f_j(1) \geq f_j(0) \\ \bar{f}_j(1) = f_j(0) - f_j(1) \text{ and } \bar{x}_j = 1 - x_j, & \text{otherwise.} \end{array}$$

In addition, recent work by Hooker (to appear) shows how nonseparable objective functions may be represented by BDDs.

5. Exact and Relaxed BDDs

If $\text{Sol}(B)$ is equal to the feasible set of an optimization problem, we will say that B is an *exact BDD* for the problem. If $\text{Sol}(B)$ is a superset of the feasible set, B is a *relaxed BDD* for the problem. We will construct *limited-width* relaxed BDDs by requiring $\omega(B)$ to be at most some pre-set maximum width W.

Figure 1(c) shows a relaxed BDD B' of width 1 for the independent set problem instance of Fig. 1(a). B' represents 21 vertex sets, including the 10 independent sets. The length of a longest path in B' is therefore an upper bound on the optimal value of the original problem instance. If, again, for all vertices v_j weight $w_j = 1$, the longest path length is 3, which provides an upper bound on the maximum cardinality 2 of an independent set.

6. Exact BDD Compilation

We now describe an algorithm that builds an exact reduced BDD for the independent set problem. Similar algorithms can be designed for any optimization problem on binary variables by associating a suitable state with each node (Hoda et al. 2010). Choosing the state variable can be viewed as analogous to formulating a model for the LP relaxation, because it allows the BDD to reflect the problem at hand.

Starting with the root r, the procedure constructs the BDD $B = (U, A)$ layer by layer, selecting a graph vertex for each layer and associating a state with each node. We define the state as follows. Using a slight abuse of notation, let $\text{Sol}(B)$ be the set of independent sets represented by B (rather than the corresponding set of tuples x). Thus, in particular, $\text{Sol}(B_{ru})$ is the set of independent sets defined by paths from r to u. Let the *neighborhood* $N(T)$ of a vertex set T be the set of vertices adjacent to vertices in T, where

by convention $T \subseteq N(T)$. The *state* $s(u)$ of node u is the set of vertices that can be added to any of the independent sets defined by paths from r to u. Thus

$$s(u) = \{v_{\ell(u)}, \ldots, v_n\} \setminus \bigcup_{T \in \mathrm{Sol}(B_{ru})} N(T).$$

In an exact BDD, all paths to a given node u define partial assignments to x that have the same feasible completions. So $s(u) = \{v_{\ell(u)}, \ldots, v_n\} \setminus N(T)$ for any $T \in \mathrm{Sol}(B_{ru})$. In addition, no two nodes on the same layer of an exact reduced BDD have the same feasible completions. So we have the following:

Lemma 1 *An exact BDD for G is reduced if and only if $s(u) \neq s(u')$ for any two nodes u, u' on the same layer of the BDD.*

The exact BDD compilation is stated in Algorithm 1. We begin by creating the root r of B, which has state $s(r) = V$ because every vertex in V is part of some independent set. We then add r to a pool P of nodes that have not yet been placed on some layer. Each node $u \in P$ is stored along with its state $s(u)$ and the arcs that terminate at u.

To create layer L_j, we first select the j-th vertex v_j by means of a function `select` (step 4), which can follow a predefined order or select vertices dynamically. We let L_j contain the nodes $u \in P$ for which $v_j \in s(u)$. These are the only nodes in P that will have both outgoing arcs $a_0(u)$ and $a_1(u)$. All of the remaining nodes in P would have only an outgoing 0-arc if placed on this layer and can therefore be skipped. The nodes in L_j are removed from P, as we need only process them once.

For each node u in L_j, we create outgoing arcs $a_0(u)$ and $a_1(u)$ as follows. Node $b_0(u)$ (i.e., the node at the opposite end of $a_0(u)$) has state $s_0 = s(u) \backslash \{v_j\}$, and node $b_1(u)$ has state $s_1 = s(u) \backslash N(\{v_j\})$. To ensure that the BDD is reduced, we check whether $s_0 = s(u')$ for some node $u' \in P$, and if so let $b_0(u) = u'$. Otherwise, we create node u_0 with $s(u_0) = s_0$, let $b_0(u) = u_0$, and insert u_0 into P. If $s_0 = \emptyset$, u_0 is the terminal node t. Arc $a_1(u)$ is treated similarly. After the last iteration, P will contain exactly one node with state $\emptyset$, and it becomes the terminal node t of B.

We now show this algorithm returns the exact BDD.

Theorem 1 *For any graph $G = (V, E)$, Algorithm 1 generates a reduced exact BDD for the independent set problem on G.*

Algorithm 1 Exact BDD Compilation

1: Create node r with $s(r) = V$
2: Let $P = \{r\}$ and $R = V$
3: **for** $j = 1$ to n **do**
4: $\quad v_j = \text{select}(R, P)$
5: $\quad R \leftarrow R \backslash \{v_j\}$
6: $\quad L_j = \{u \in P : v_j \in s(u)\}$
7: $\quad P \leftarrow P \backslash L_j$
8: $\quad$ **for all** $u \in L_j$ **do**
9: $\quad\quad s_0 := s(u) \backslash \{v_j\}$, $s_1 := s(u) \backslash N(v_j)$
10: $\quad\quad$ **if** $\exists\, u' \in P$ with $s(u') = s_0$ **then**
11: $\quad\quad\quad a_0(u) = (u, u')$
12: $\quad\quad$ **else**
13: $\quad\quad\quad$ create node u_0 with $s(u_0) = s_0$ ($u_0 = t$ if $s_0 = \emptyset$)
14: $\quad\quad\quad a_0(u) = (u, u_0)$
15: $\quad\quad\quad P \leftarrow P \cup \{u_0\}$
16: $\quad\quad$ **if** $\exists\, u' \in P$ with $s(u') = s_1$ **then**
17: $\quad\quad\quad a_1(u) = (u, u')$
18: $\quad\quad$ **else**
19: $\quad\quad\quad$ create node u_1 with $s(u_1) = s_1$ ($u_1 = t$ if $s_0 = \emptyset$)
20: $\quad\quad\quad a_1(u) = (u, u_1)$
21: $\quad\quad\quad P \leftarrow P \cup \{u_1\}$
22: Let t be the remaining node in P and set $L_{n+1} = \{t\}$

Proof. Let $\text{Ind}(G)$ be the collection of independent sets of G. We wish to show that if B is the BDD created by Algorithm 1, $\text{Sol}(B) = \text{Ind}(G)$. We proceed by induction on $n = |V|$.

First, suppose $n = 1$, and let G consist of a single vertex v. B consists of two nodes, r and t, and two arcs $a_0(r)$ and $a_1(r)$, both directed from r to t. Therefore, $\text{Sol}(B) = \{\emptyset, v\} = \text{Ind}(G)$. Moreover, this BDD is trivially reduced.

For the induction hypothesis, suppose that Algorithm 1 creates a reduced exact BDD for any graph on fewer than n (≥ 2) vertices. Let G be a graph on n vertices. Suppose the `select` function in Step 4 returns vertices in the order $v_1, \ldots, v_n$. Let $G_0 = (V_0, E_0)$ be the subgraph of G induced by vertex set $V \backslash \{v_1\}$, and $G_1 = (V_1, E_1)$ the subgraph induced by $V \backslash N(v_1)$. Then $\text{Ind}(G) = \text{Ind}(G_0) \cup \{T \cup \{v_1\} \mid T \in \text{Ind}(G_1)\}$, since each indepen-

dent set either excludes v_1 (whereupon it appears in $\mathrm{Ind}(G_0)$) or includes v_1 (whereupon it appears as the union of $\{v_1\}$ with a set in $\mathrm{Ind}(G_1)$).

Let B be the BDD returned by the algorithm for G. By construction, $s(b_0(r)) = V_0$ and $s(b_1(r)) = V_1$. Let B_0 be the BDD that the algorithm creates for G_0, and similarly for B_1. We observe as follows that $B_0 = B_{b_0(r)t}$ and $B_1 = B_{b_1(r)t}$. The root r_0 of B_0 has $s(r_0) = V_0$, the same state as node $b_0(r)$ in B. But the successor nodes created by the algorithm for r_0 and $b_0(r)$ depend entirely on the state and are therefore identical in B_0 and B, respectively. Moreover, the states of the successor nodes depend entirely on the state of the parent and which branch is taken. Thus the successor nodes have the same states in B_0 as in B. If we apply this reasoning recursively, we obtain $B_0 = B_{b_0(r)t}$. A parallel argument shows that $B_1 = B_{b_1(r)t}$. Now

$$\begin{aligned}
\mathrm{Sol}(B) &= \mathrm{Sol}(B_{b_0(r)t}) \cup \{T \cup \{v_1\} \mid T \in \mathrm{Sol}(B_{b_1(r)t})\} \\
&= \mathrm{Sol}(B_0) \cup \{T \cup \{v_1\} \mid T \in \mathrm{Sol}(B_1)\} \\
&= \mathrm{Ind}(G_0) \cup \{T \cup \{v_1\} \mid T \in \mathrm{Ind}(G_1)\} \\
&= \mathrm{Ind}(G)
\end{aligned}$$

as claimed, where the third equation is due to the inductive hypothesis. Furthermore, since all nodes with the same state are merged, Lemma 1 implies that B is reduced. □

To analyze the time complexity of Algorithm 1, we assume that the `select` function (Step 4) is "polynomial" in the sense that its running time is at worst proportional to $|V|$ or the number of BDD nodes created so far, whichever is greater.

Lemma 2 *If the* `select` *function is polynomial, then the time complexity of Algorithm 1 is polynomial in the size of the reduced exact BDD $B = (U, A)$ constructed by the algorithm.*

Proof. We observe that an arc of B is never rechecked again once it was created in one of the Steps 11, 14, 17, or 20. Hence, the complexity of the algorithm is dominated by the `select` function or the constructive operations required when creating the out-arcs of a node removed from the pool P. The `select` function is clearly polynomial in $|U|$ and $|V|$. The constructive operations consist of creating a new state (Step 9) and inserting or searching in the node pool (Steps 10, 15, 16, and 21), which can be implemented in $O(|V|)$. Since every node has exactly two outgoing arcs (i.e., $|A| = 2|U|$), the resulting worst-case complexity is $O(|U|\,|V|)$, and the lemma follows. □

7. Relaxed BDDs

Limited-width relaxed BDDs allow us to represent an over-approximation of the family of independent sets of a graph, and thus obtain an upper bound on the optimal value of the independent set problem.

We propose a novel top-down compilation method for constructing relaxed BDDs. The procedure modifies Algorithm 1 by forcing nodes to be merged when a particular layer exceeds a pre-set maximum width W. This modification is given in Algorithm 2, which is to be inserted immediately after line 7 in Algorithm 1.

The procedure is as follows. We begin by checking if $\omega_j > W$, which indicates that the width of layer L_j exceeds W. If so, we select a subset M of L_j using function `node_select` in Step 2, which ensures that $2 \leq |M| \leq \omega_j - W$. The set M represents the nodes to be merged so that the desired width is met. Various heuristics for selecting M are discussed in Section 8.

The state of the new node that results from the merge, s_{new}, must be such that no feasible independent set is lost in further iterations of the algorithm. As will be established by Theorem 2, it suffices to let s_{new} be the union of the states associated with the nodes in M (Step 3). Once s_{new} is created, we search for some node $u' \in L_j$ such that $s(u') = s_{new}$. If u' exists, then by Lemma 1 we are only required to direct the incoming arcs of the nodes in M to u', as presented in Algorithm 3. Otherwise, we create a new node $\hat{u}$ with $s(\hat{u}) = s_{new}$ and add it to L_j.

In each iteration of the *while* loop in Algorithm 2, we decrease the size of L_j by at least $|M| - 1$. Thus, after at most $\omega_j - W$ iterations, the layer L_j will have width no greater than W. The modified Algorithm 1 hence yields a limited-width W BDD, i.e. $\omega(B) \leq W$.

The correctness of Algorithm 2 is proved by showing that every r-t path of the exact BDD remains after merging operations.

Theorem 2 *For any graph $G = (V, E)$, Algorithm 1 modified by adding Algorithm 2 after line 7 generates a relaxed BDD.*

Proof. We will use the notation B_u for the BDD consisting of all r–t paths in B that pass through u. Thus

$$\mathrm{Sol}(B_u) = \{V_1 \cup V_2 \mid V_1 \in \mathrm{Sol}(B_{ru}),\ V_2 \in \mathrm{Sol}(B_{ut})\} \tag{1}$$

It suffices to show that each iteration of the while-loop yields a relaxed BDD if it begins with a relaxed BDD. Thus we show that if B is a relaxed (or

Algorithm 2 Node merger for obtaining a relaxed BDD.
Insert immediately after line 7 of Algorithm 1.

```
1: while ω_j > W do
2:    M := node_select(L_j)  // where 2 ≤ |M| ≤ ω_j − W
3:    s_new := ⋃_{u∈M} s(u)
4:    L_j ← L_j \ M
5:    if ∃u' ∈ L_j with s(u') = s_new then
6:       merge(M, u')
7:    else
8:       Create node û with s(û) = s_new
9:       merge(M, û)
10:      L_j = L_j ∪ {û}
```

Algorithm 3 merge(M, u')

```
1: for all u ∈ M do
2:    for all arcs a_0(w) with b_0(w) = u do
3:       b_0(w) ← u'
4:    for all arcs a_1(w) with b_1(w) = u do
5:       b_1(w) ← u'
```

exact) BDD, then the BDD $\hat{B}$ that results from merging the nodes in M satisfies $\mathrm{Sol}(B) \subseteq \mathrm{Sol}(\hat{B})$. Here M is any proper subset of L_j for an arbitrary $j \in \{2, \ldots, n-1\}$.

Let $M = \{u_1, \ldots, u_k\}$ be the nodes to be merged into $\hat{u}$. Also, let $\bar{B}$ be the BDD consisting of all r–t paths in B that do *not* include any of the nodes u_i. Then

$$\mathrm{Sol}(B) = \mathrm{Sol}(\bar{B}) \cup \bigcup_{i=1}^{k} \mathrm{Sol}(B_{u_i})$$

The merge procedure has no effect on $\mathrm{Sol}(\bar{B})$. Hence it remains to show that

$$\bigcup_{i=1}^{k} \mathrm{Sol}(B_{u_i}) \subseteq \mathrm{Sol}(\hat{B}_{\hat{u}})$$

But we can write

$$\begin{aligned}
\bigcup_{i=1}^{k} \mathrm{Sol}(B_{u_i}) &= \bigcup_{i=1}^{k} \{V_1 \cup V_2 \mid V_1 \in \mathrm{Sol}(B_{ru_i}),\ V_2 \in \mathrm{Sol}(B_{u_it})\} \\
&= \left\{ V_1 \cup V_2 \;\middle|\; V_1 \in \bigcup_{i=1}^{k} \mathrm{Sol}(B_{ru_i}),\ V_2 \in \bigcup_{i=1}^{k} \mathrm{Sol}(B_{u_it}) \right\} \\
&= \left\{ V_1 \cup V_2 \;\middle|\; V_1 \in \mathrm{Sol}(\hat{B}_{r\hat{u}}),\ V_2 \in \bigcup_{i=1}^{k} \mathrm{Sol}(B_{u_it}) \right\} \\
&\subseteq \left\{ V_1 \cup V_2 \;\middle|\; V_1 \in \mathrm{Sol}(\hat{B}_{r\hat{u}}),\ V_2 \in \mathrm{Sol}(\hat{B}_{\hat{u}t}) \right\} \\
&= \mathrm{Sol}(\hat{B}_{\hat{u}})
\end{aligned}$$

The first and last equations are due to (1). The third equation is due to $\bigcup_i \mathrm{Sol}(B_{ru_i}) = \mathrm{Sol}(\hat{B}_{r\hat{u}})$, which follows from the fact that $\hat{u}$ receives precisely the paths received by the u_is before the merge. The fourth line is due to $\bigcup_i \mathrm{Sol}(B_{u_it}) \subseteq \mathrm{Sol}(\hat{B}_{\hat{u}t})$. This follows from the facts that (a) $\mathrm{Sol}(B_{u_it})$ contains the independent sets in the subgraph of G induced by $s(u_i)$; (b) $\mathrm{Sol}(\hat{B}_{\hat{u}t})$ contains the independent sets in the subgraph induced by $s(\hat{u})$; and (c) $s(u_i) \subseteq s(\hat{u})$ for all i. □

The time complexity of Algorithm 2 is highly dependent on the `node_select` function and on the number of nodes to be merged. Once a subset M of nodes has been chosen, taking the union of the states (Step 3) has a time complexity of $O(|M||V|)$, and Algorithm 3 has a worst-case time complexity of $O(W|M|)$ by supposing that every node in M is adjacent to as many as W nodes located in previous layers. Hence, if k is the number of nodes to be merged, the complexity of Algorithm 2 is $O(H(k) + |M||V| + W|M|)$ per iteration of the *while* loop in Step 1, where $H(k)$ is the complexity of the node selection heuristic (`node_select`) for a given k. The number of iterations depends on the size of the selected node set. For example, if $|M|$ is always 2, then at most $W - k$ iterations are required (if none of the newly defined states appeared in L_j previously). The time complexity for the complete relaxation procedure is given by the following lemma.

Lemma 3 *Let S be the time complexity of selecting the next variable (`select` function in Step 4 of Algorithm 1), and let $R(k)$ be the time complexity of*

Algorithm 2. The worst-case time complexity of Algorithm 1 modified with the procedure in Algorithm 2 is given by $O(n(S + R(nW) + W|V|))$.

Proof. If k nodes are removed from the pool in Step 6 of Algorithm 1, then the merging procedure in Algorithm 2 ensures that at most $2\min\{k, W\}$ new nodes are added back to the pool. Thus, at each iteration the pool can be increased by at most W nodes. Since n iterations in the worst case are required for the complete compilation, the pool can have at most nW nodes.

Suppose now nW nodes are removed from the pool (Step 6 of Algorithm 1) at a particular iteration. These nodes are first merged so that the maximum width W is met (Algorithm 2), and then new nodes or arcs are created according to the result of the merge. The time complexity for the first operation is $R(nW)$, which yields a new layer with at most W nodes. For the second operation, we observe as in Lemma 2 that creating a new state or searching in the pool size can be implemented in time $O(|V|)$; hence, the second operation has a worst-case time complexity of $O(W|V|)$.

This implies that the time required per iteration is $O(S+R(nW)+W|V|)$, yielding a time complexity of $O(n(S + R(nW) + W|V|))$ for the modified procedure. □

8. 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.

In this section we present a number of possible heuristics for selecting nodes. This refers to how the subsets M are chosen on line 2 in Algorithm 2. 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.

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.

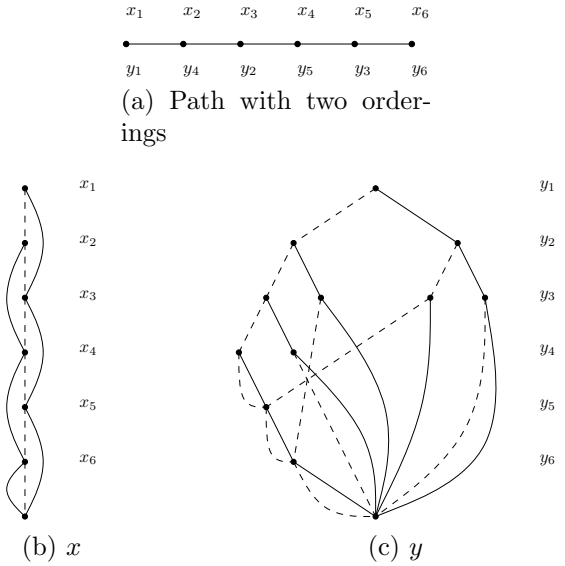
minSize: Sort nodes in L_j in decreasing order of their corresponding state sizes and merge the first 2 nodes until $|L_j| \leq W$. This heuristic merges nodes that have the largest number of vertices in their associated states. Because larger vertex sets are likely to have more vertices in common, the heuristic tends to merge nodes that represent similar regions of the solution space.

9. Variable Ordering

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 (Ebendt et al. 2003, Bollig and Wegener 1996). 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. For example, Figure 2a shows a path on 6 vertices with two different orderings given by $x_1, \ldots, x_6$ and $y_1, \ldots, y_6$. In Figure 2b we see that the vertex ordering $x_1, \ldots, x_6$ yields an exact BDD with width 1, while in Figure 2c 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.

In the remainder of this section we describe classes of graphs for which an appropriate ordering of the vertices leads to a bound the width of the exact BDD. In addition, we provide a set of orderings based on maximal path decompositions that yield exact reduced BDDs in which the width of layer L_j is bounded by the $(j+1)$-st Fibonacci number for any graph. Based on this analysis, we describe various heuristic orderings for reduced BDDs, on the assumption that an ordering that results in a small-width exact reduced BDD also results in a relaxed BDD that yields a strong bound on the objective function.



(a) Path with two orderings

(b) x (c) y

Figure 2: Path with two different variable orderings.

9.1 Exact BDD Orderings

Here we present orderings of vertices for interval graphs, trees, and general graphs for which we can bound the width, and therefore the size, of the exact reduced BDD.

We first consider interval graphs; that is, graphs that are isomorphic to the intersection graph of a multiset of intervals on the real line. Such graphs have variable orderings $v_1, \ldots v_n$ for which each vertex v_i is adjacent to the set of vertices $v_{a_i}, v_{a_i+1}, \ldots v_{i-1}, v_{i+1}, \ldots, v_{b_i}$ for some a_i, b_i. We call such an ordering an *interval ordering* for G. Note that paths and cliques, for example, are contained in this class of graphs.

Theorem 3 *For any interval graph, an interval ordering $v_1, \ldots, v_n$ yields an exact reduced BDD with width 1.*

Proof. Let $T_k = \{v_k, \ldots, v_n\}$. We first show by induction that for any interval ordering $v_1, \ldots, v_n$, there is is k such that $s(u) = V_k$ for for all $u \in P$ throughout the entire execution of Algorithm 1.

At the start of the algorithm, $P = \emptyset, L_1 = \{r\}$ with $s(r) = T_1$. Starting from r, we have $s_0 = s(r) \setminus \{v_1\} = T_2$, and $s_1 = s(r) \setminus N(v_1) = T_{b_1+1}$. Therefore, at the end of iteration $j = 1$, P contains two nodes with states T_2 and T_{b_1+1}.

Now fix an arbitrary $j < n$ and assume for the induction hypothesis that, at the start of iteration j, each node $u \in P$ has $s(u) = T_k$. Note that $k \geq j$, since at each iteration we always eliminate v_j for each state (if it appears). Then $|L_j| = 1$, because there can be at most one node $u' \in P$ with $v_j \in s(u')$, and the only T_k with $k \geq j$ that contains v_j is T_j. Starting from u', we have $s_0 = V_j \setminus \{v_j\} = T_{j+1}$ and $s_1 = V_j \setminus N(v_j) = T_{b_j+1}$. We therefore add at most two nodes to P at the end of iteration j, each with a state of form T_k for some k. This proves the claim.

We conclude that $s(u) = T_k$ for all $u \in P$. For any j, we have $|L_j| = 1$, because there is at most one node with $v_j \in s(u)$. Therefore, $\omega(B) = 1$. $\square$

We now prove a width bound for trees.

Theorem 4 *For any tree with $n \geq 2$ vertices, there exists an ordering of the vertices that yields an exact reduced BDD with width at most $n/2$.*

The proof will use a lemma demonstrated in Jordan (1869).

Lemma 4 *In any tree there exists a vertex v for which the connected components created upon deleting v from the tree contain at most $n/2$ vertices.*

Proof of Theorem 4. We proceed by induction on n. For $n = 2$, any tree is an interval graph, and by Theorem 3 there exists an ordering of the vertices that yields an exact reduced BDD with width 1.

For the inductive hypothesis, we suppose that for for any tree T with $n' < n$ vertices, there exists an ordering of the vertices in T for which the exact reduced BDD has width at most $n'/2$. Let T be any tree with n vertices. Let v be a cut vertex satisfying the conditions of Lemma 4. Each connected component $T_i = (V_i, E_i)$, $i = 1, \ldots, k$, created upon deleting v from T is a tree. By induction, for each i, there is an ordering of the vertices in V_i for which the exact reduced BDD for T_i has width at most $\frac{|V_i|}{2} \leq \frac{n}{4}$. Let $v_1^i, \ldots, v_{|V_i|}^i$ be such an ordering and $B_i = (U_i, A_i)$ be the exact reduced BDD for T_i with this ordering.

Consider the ordering $v_1^1, \ldots, v_{|V_1|}^1, v_1^2, \ldots, v_{|V_k|}^k, v$ of the vertices in T; i.e., we order the vertices by the component orderings that yield an exact reduced BDD with width at most $\frac{n}{4}$, followed by the cut vertex v. We now show that using this ordering, the exact reduced BDD $B = (U, A)$ for T has width at most $2 \cdot \frac{n}{4} = \frac{n}{2}$, finishing the proof.

Fix $j, 1 \leq j \leq n-1$, and let v_ℓ^i be the jth vertex in the ordering for T. We claim that using this ordering, for each vertex $u \in L_j$ of B, there exists a $w \in L_\ell^i$, the ℓth layer of B_i, for which

$$s(u) = \begin{cases} s(w) \bigcup (V_{i+1} \cup \cdots \cup V_k) \\ s(w) \bigcup (V_{i+1} \cup \cdots \cup V_k) \setminus \{v\}. \end{cases}$$

Consider the BDD B_{ru}. For every set $W \in \mathrm{Sol}(B_{ru})$ we have that $v_\ell^i \in s(u)$ and

$$\begin{aligned} s(u) &= (v_j \cup v_{j+1} \cup \cdots \cup v_n) \setminus N(W) \\ &= \left(\left\{ v_\ell^i \cup v_{\ell+1}^i \cup \cdots \cup v_{|V_i|}^i \right\} \cup \left\{ V^{i+1} \cup \cdots \cup V^k \right\} \right) \setminus N(W) \\ &= \left(\left\{ v_\ell^i \cup v_{\ell+1}^i \cup \cdots \cup v_{|V_i|}^i \right\} \setminus N(W) \right) \cup \left\{ V^{i+1} \cup \cdots \cup V^k \right\}, \end{aligned}$$

where the last equality follows because the nodes in W are not adjacent to any vertex in the remaining components.

Now, consider the set $W[V^i] = W \cap V^i$. This must be an independent set in the graph $T[V^i]$, the graph induced by vertex set V^i. In addition, $v_\ell^i \in V^i \setminus N(W[V^i])$ is in $T[V^i]$ because this vertex also appears in $s(u)$. Therefore, there is a node $w \in L_\ell^i$ with state $s(w) = \left\{ v_\ell^i \cup v_{\ell+1}^i \cup \cdots \cup v_{|V_i|}^i \right\} \setminus N(W[V^i])$. In the entire graph T, $N(W)$ contains exactly the vertices in $T[V^i]$ that are in $N(W[V^i])$ and possibly vertex v, because v is the only vertex, besides those vertices in V^i, that can be adjacent to any vertex in V^i, as desired. □

Finally, we prove a bound for general graphs.

Theorem 5 *Given any graph, there exists an ordering of the vertices that yields an exact reduced BDD $B = (U, A)$ with width equal to at most the jth Fibonaci number Fib_{j+1}.*

Proof. Given graph $G = (V, E)$, we define a *maximal path decomposion ordering* of V as an ordered partition of the vertex set $V = V^1 \cup \cdots \cup V^k$ together with an ordering $v_1^i, \ldots, v_{|V^i|}^i$ of the vertices in each partition V^i for which

$$\begin{aligned} &\left(v_j^i, v_{j+1}^i\right) \in E && \text{for } i = 1, \ldots, k, j = 1, \ldots, |V^k| - 1 \\ &N\left(v_{|V^i|}^i\right) \subseteq V^1 \cup \cdots \cup V^i && \text{for } i = 1, \ldots, k. \end{aligned}$$

Thus each partition is covered by a path whose last vertex is independent of all vertices in the remaining partitions.

We show that for any maximal path decomposition ordering, the exact reduced BDD B will have $\omega_j \leq \text{Fib}_{j+1}$. Let P_j be the pool of nodes in Algorithm 1 before line 4 in iteration j. We will show that $|P_j| \leq \text{Fib}_{j+1}$, which implies that $\omega_j = |L_j| \leq |P_j| \leq \text{Fib}_{j+1}$, as desired.

We first consider the case with $k = 1$; i.e., the graph contains a hamiltonian path. Let $v_1, \ldots, v_n$ be a hamiltonian path in G and the maximal path decomposition ordering we use to create an exact reduced BDD for G. We proceed by induction.

We first note that $P_1 = \{r\}$, so that $|P_1| = 1$. Now $L_1 = \{r\}$, and with $u = r$ in the algorithm, we have $s_0 = T_2$ and $s_1 = T_2 \setminus N(v_1)$. Since $(v_1, v_2) \in E$, these two states are different, so that $P_2 = \{u_1^2, u_2^2\}$ with $s(u_1^2) = T_2$ and $s(u_2^2) = T_2 \setminus N(v_1)$. However, $v_2 \notin s(u_2^2)$ because $(v_1, v_2) \in E$, so that $L_2 = \{u_1^2\}$. Node u_1^2 can result in the addition of at most two more nodes to P_2 in when creating P_3, one with state s_0 and one with state s_1. Therefore, $|P_3| \leq 3 \leq |P_2| + |P_1| = 2 + 1 = 3 = \text{Fib}_4$, as desired.

For the inductive hypothesis, suppose $|P_j| \leq \text{Fib}_{j+1}$ for $j \leq j'$. We seek to show that

$$|P_{j'+1}| \leq \text{Fib}_{j'+2} \tag{2}$$

Consider the partition of the nodes in $P_{j'}$ into $X \cup Y$, where a node $u \in P_{j'}$ is in X if there exists a node $u' \in L_{j'-1}$ for which $b_1(u') = u$; i.e., there is a 1-arc ending at u directed out of a node in $L_{j'-1}$. All other nodes are in Y. We make three observations.

1. $|Y| \leq |P_{j'-1}|$
   The nodes in $P_{j'-1}$ can be partitioned into $L_{j'-1} \cup \bar{L}_{j'-1}$. All nodes in $P_{j'}$ either arise from a 0-arc or 1-arc directed out of $L_{j'-1}$ or are copies of the nodes in $\bar{L}_{j'-1}$ (these may be combined because their associated states may coincide). Only the nodes arising from 1-arcs directed out of nodes in $L_{j'-1}$ are in X, and the remaining nodes are in Y. There are at most $|L_{j'-1}| + |\bar{L}_{j'-1}|$ nodes in Y, including at most $|L_{j'-1}|$ from 0-arcs and at most $|\bar{L}_{j'-1}|$ copies of nodes from $P_{j'-1}$. Therefore, $|Y| \leq |L_{j'-1}| + |\bar{L}_{j'-1}| = |P_{j'-1}|$.

2. $|L_{j'}| \leq |Y|$
   We have that $L_{j'} \subseteq P_{j'}$. However, any node $u \in X$ must have $v_{j'} \notin s(u)$ because $(v_{j'-1}, v_{j'}) \in E$. Therefore $L_{j'} \subseteq P_{j'}$ and $|L_{j'}| \leq |Y|$.

3. $|P_{j'+1}| \leq 2|L_{j'}| + (|P_{j'}| - |L_{j'}|) = |P_{j'}| + |L_{j'}|$
   As in $P_{j'}$, the nodes in $P_{j'+1}$ arise from a 0-arc or 1-arc directed out of $L_{j'}$ or are copies of the nodes in $\bar{L}_{j'}$ (these may be combined because their associated states may coincide). So each node in $L_{j'}$ gives rise to at most two nodes that are inserted $P_{j'+1}$, and each node in $\bar{L}_{j'}$ contributes at most one node to $P_{j'+1}$. The inequality follows.

Putting all three observations together, we get

$$|P_{j'+1}| \leq |P_{j'}| + |L_{j'}| \leq |P_{j'}| + |Y| \leq |P_{j'}| + |P_{j'-1}|,$$

We therefore have by induction that $|P_{j'+1}| \leq \text{Fib}_{j'+1} + \text{Fib}_{j'} = \text{Fib}_{j'+2}$, proving (2) for $k = 1$.

Now let $k > 1$. From above, we know that $\left|P_{v^1_{|V^1|}}\right| \leq \text{Fib}_{|V^1|+1}$. We first show that

$$\left|P_{v^1_{|V^1|+1}}\right| \leq \left|P_{v^1_{|V^1|}}\right| \tag{3}$$

Take any node $u \in P_{v^1_{|V^1|}}$. If $v^1_{|V^1|} \notin s(u)$ then this node is reproduced in $P_{v^1_{|V^1|+1}}$. If $v^1_{|V^1|} \in s(u)$, then $s_0 = s_1$. This is because $v^1_{|V^1|}$ is independent of all nodes in the remainder of the graph. Therefore, eliminating $v^1_{|V^1|}$ or eliminating $v^1_{|V^1|} \cup N\left(v^1_{|V^1|}\right)$ from $s(u)$ corresponds to the the set of vertices in G. This may coincide with the state of some node that originally did not have $v^1_{|V^1|}$ in its state, but in either case at most one new node is added to $P_{v^1_{|V^1|+1}}$. We therefore have (3). This in turn implies that $\left|P_{v^1_{|V^1|+1}}\right| \leq \left|P_{v^1_{|V^1|}}\right| \leq \text{Fib}_{|V^1|+1} \leq \text{Fib}_{|V^1|+2}$, as desired. In addition, since consecutive P_j's differ in size by at most a factor of 2,

$$\left|P_{v^1_{|V^1|}+2}\right| \leq 2\left|P_{v^1_{|V^1|}+1}\right| \leq 2\left|P_{v^1_{|V^1|}}\right| \leq 2\text{Fib}_{|V^1|+1} \leq \text{Fib}_{|V^1|+1},$$

as desired.

Now, since the vertices in indices $v^1_{|V^1|} + 1$ and $v^1_{|V^1|} + 2$ are v^2_1 and v^2_2, respectively, and their corresponding P_j's are bounded by the desired Fibonacci numbers, we can apply the reasoning from the proof of $k = 1$ to bound the sizes of the P_j's until the end of set V^2, and by induction bound the remaining P_j's. □

9.2 Relaxed BDD Orderings

The orderings in Section 9.1 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 above, 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.

random: Randomly select some vertex that has yet to be chosen. We suggest this vertex selection not only as a stand alone variable ordering heuristic, but also as a heuristic that may be mixed with any of the following heuristics for the purpose of generating several relaxations.

minState: Select the next vertex v_j as the vertex appearing in the fewest number of states in P. This selection 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 mentioned above, it was shown in Bergman et al. (2012) that a Maximal Path Decomposition of the vertices in a graph yields an ordering that 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 also limiting the number of merging operations necessary to build the BDD.

10. Computational Experiments

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 an 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++.

10.1 Merging Heuristics

We tested the three merging heuristics presented in Section 8 on the `random` instance set. We set a maximum width of $W = 10$ and used variable ordering heuristic `MPD`. Figure 3 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.

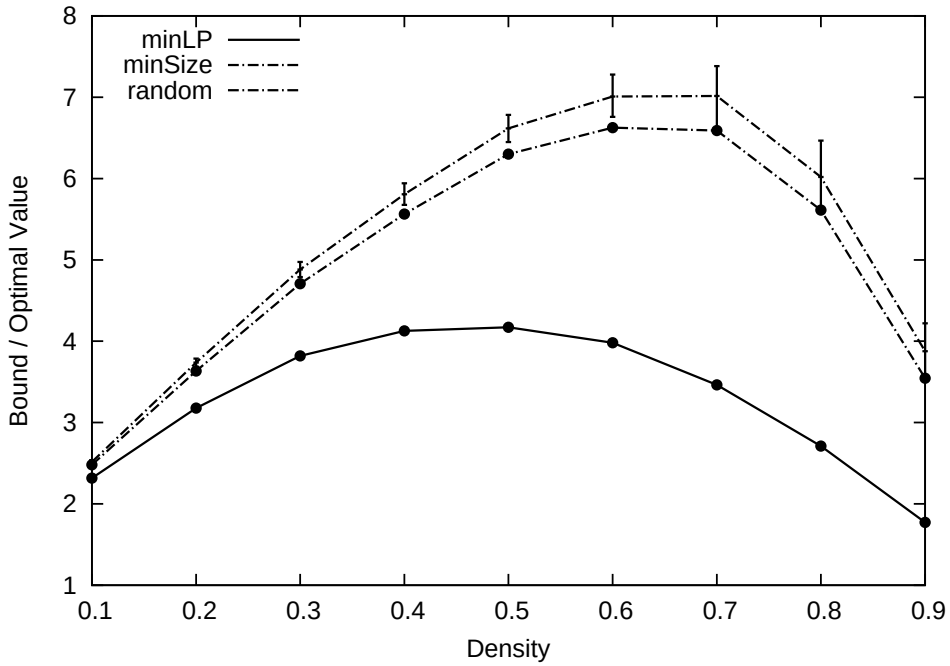


Figure 3: Bound quality vs. graph density for each merging heuristic, using the `random` instance set with `MPD` ordering and maximum BDD width 10. Each data point represents an average over 20 problem instances. The vertical line segments indicate the range obtained in 5 trials of the random heuristic.

10.2 Variable Ordering Heuristics

We tested the three variable ordering heuristics presented in Section 9 on the `random` instance set. The results (Fig. 4) 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.

10.3 Bounds vs. Maximum BDD Width

The purpose of this experiment is to analyze the impact of maximum BDD width on the resulting bound. Figure 5 presents the results for instance

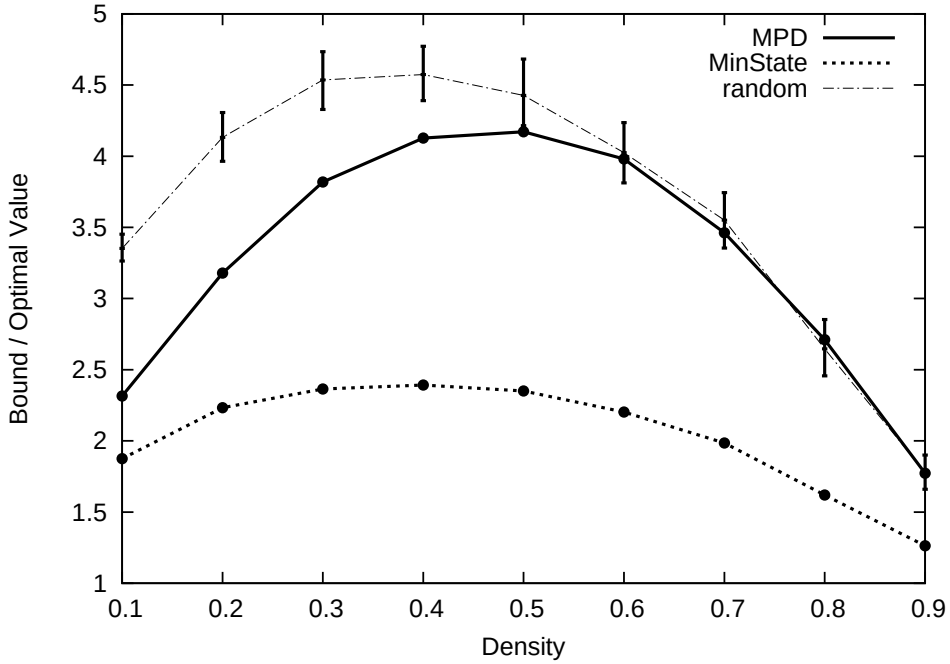


Figure 4: Bound quality vs. graph density for each variable ordering heuristic, using merge heuristic **minLP** and otherwise the same experimental setup as Fig. 3.

`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.

10.4 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 (Grötschel et al. 1993) of the maximum independent set problem, which requires computing a clique cover before the model can be formulated. We then augmented the LP relaxation with cutting planes generated at the root node by the CPLEX MILP solver.

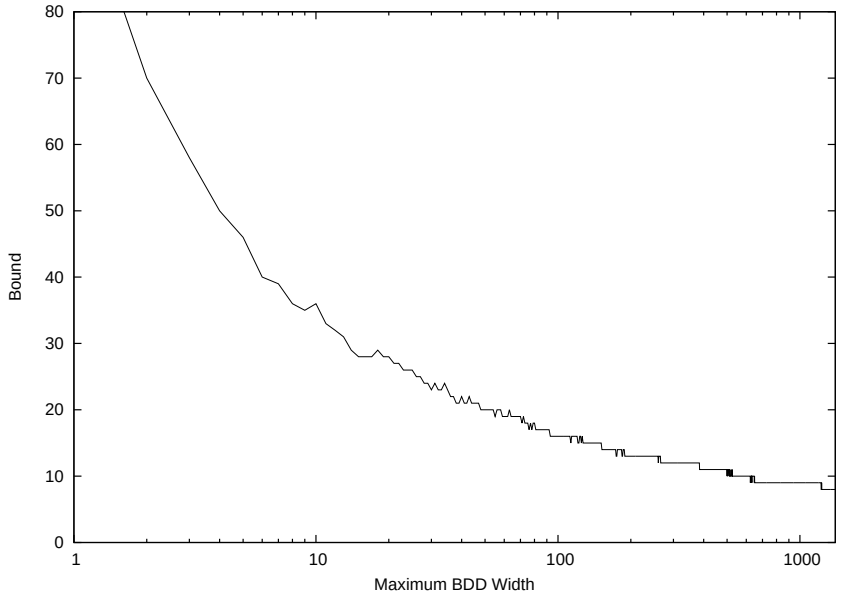


Figure 5: Relaxation bound vs. maximum BDD width for **dimacs** instance `p-hat_300-1`.

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 1 and are plotted in Fig. 6. The table displays geometric means, rather than averages, to reduce the effect of outliers. It uses shifted geometric means[1] for computation times. The computation times for LP include the time necessary to compute the clique cover, which is much less than the time required to solve the initial LP for **random** instances, and about the same as the LP solution time for **dimacs** instances.

The results show that BDDs with width as small as 100 provide bounds that, after taking means, are superior to LP bounds for all graph densities except 0.1. The computation time required is about the same overall—more for sparse instances, less for dense instances. The scatterplot in Fig. 8 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. 10. 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 2 and Fig. 7, with scatter plots in Figs. 11–13. 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 3 shows detailed results for each instance.

BDDs of width 100 provide somewhat better bounds than the LP without

[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 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**

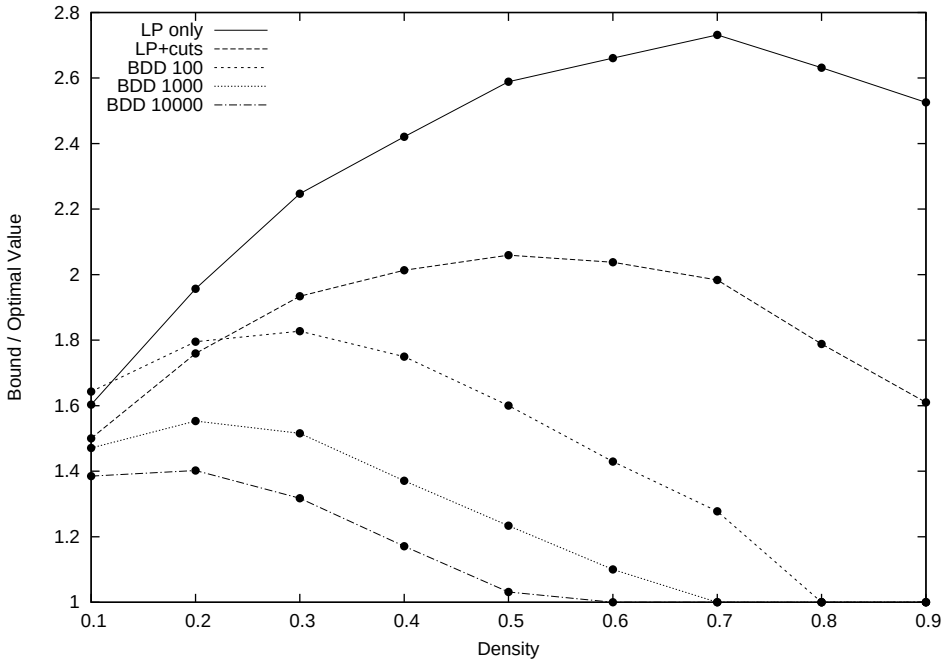


Figure 6: Bound quality vs. graph density for **random** instances, showing results for LP only, LP plus cutting planes, and BDDs with maximum width 100, 1000, and 10000. Each data point is the geometric mean of 20 instances.

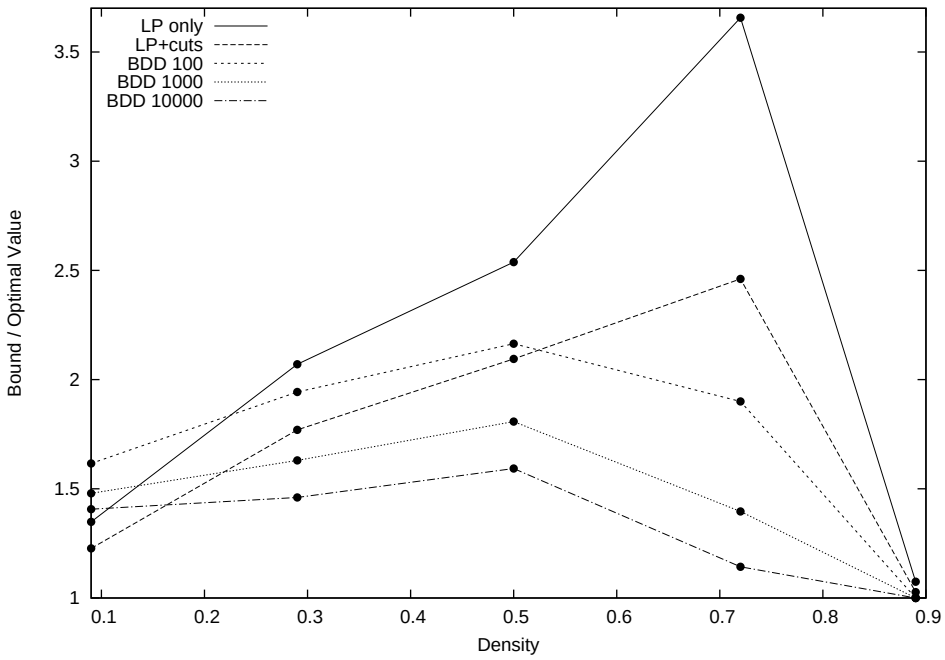


Figure 7: Bound quality vs. graph density for **dimacs** instances, showing results for LP only, LP plus cutting planes, and BDDs with maximum width 100, 1000, and 10000. Each data point is the geometric mean of instances in a density interval of width 0.2.

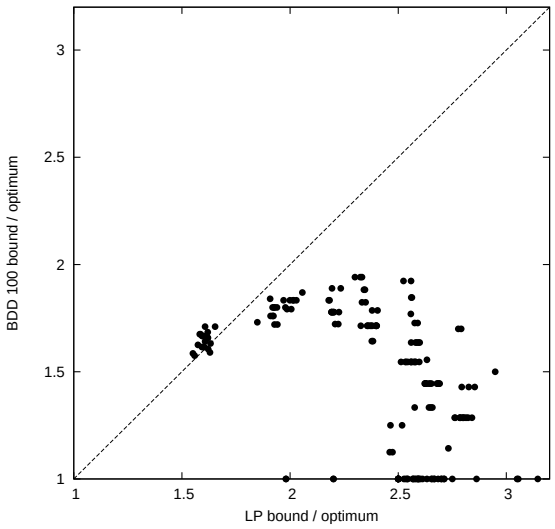


Figure 8: Bound quality for an LP relaxation (no cuts) vs. width 100 BDDs for **random** instances. Each data point represents one instance. The time required is about the same overall for the two types of bounds.

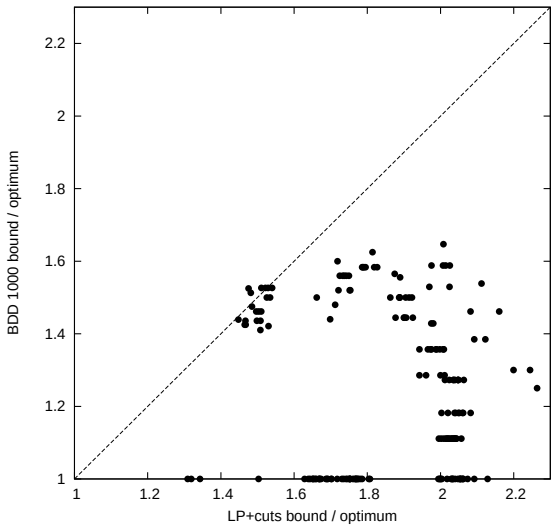


Figure 9: Bound quality for an LP relaxation with cuts vs. width 1000 BDDs for **random** instances. The BDD bounds are obtained in about 5% of the time required for the LP bounds.

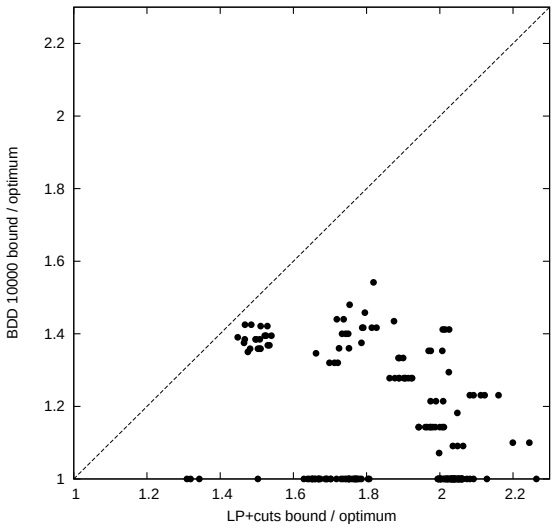


Figure 10: Bound quality for an LP relaxation with cuts vs. width 10000 BDDs for **random** instances. The BDD bounds are obtained in less time overall than the LP bounds, but somewhat more time for sparse instances.

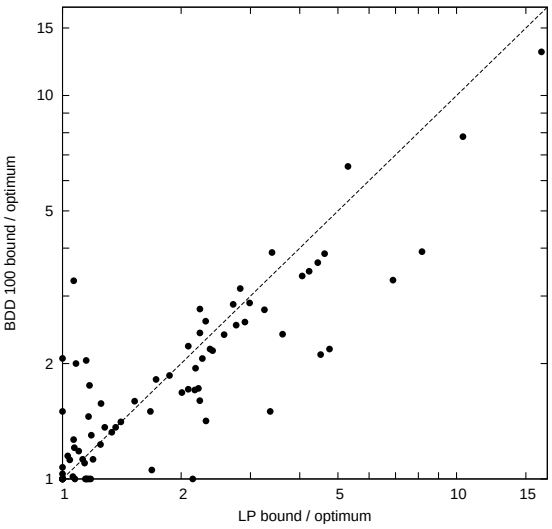


Figure 11: Bound quality for an LP relaxation (no cuts) vs. width 100 BDDs for **dimacs** instances. The BDD bounds are obtained in generally less time for all but the sparsest instances.

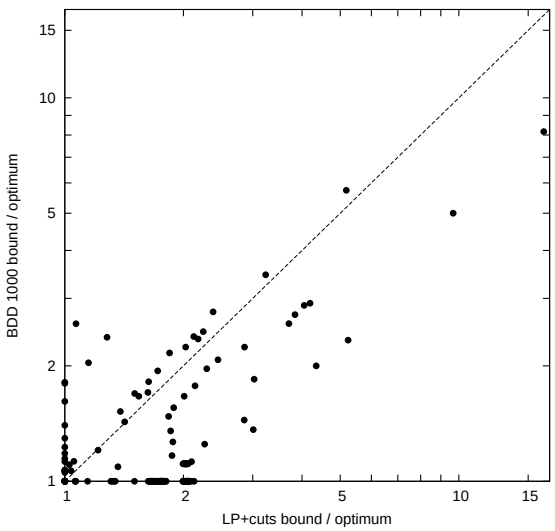


Figure 12: Bound quality for an LP relaxation with cuts vs. width 1000 BDDs for **dimacs** instances. The BDD bounds are obtained in about 15% as much time overall as the LP bounds.

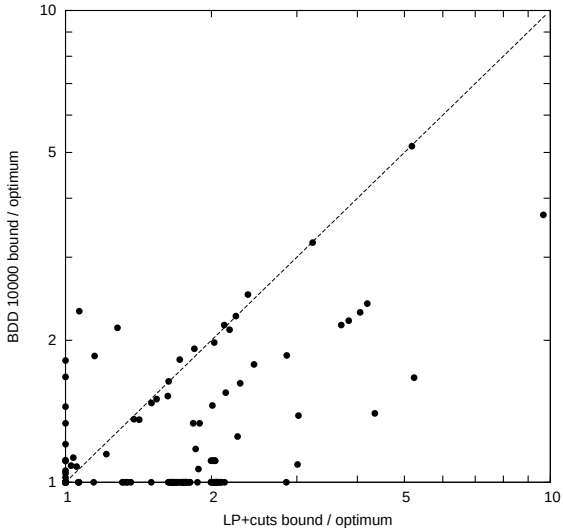


Figure 13: Bound quality for an LP relaxation with cuts vs. width 10000 BDDs for **dimacs** instances. The BDD bounds are generally obtained in less time for all but the sparsest instances.

cuts, except for the sparsest instances, and the computation time is somewhat less overall. Again, however, the more important comparison is with LP augmented by cutting planes. BDDs of width 100 are no longer superior, but increasing the width to 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.

Table 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 sec. (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**

11. Conclusions

In this paper we presented a novel method, based on binary decision diagrams (BDDs), for obtaining bounds on the optimal value of discrete optimization problems. As a test case, we applied the technique to the maximum independent set problem. We found that the BDD-based bounding procedure often yields better bounds, in less time, than a state-of-the-art mixed-integer solver obtains at the root node for a tight integer programming model.

The performance of both BDD and conventional relaxations is senstitive to the density of the graph. We found, however, that BDDs yield tighter bounds in less time, taking the geometric mean, for random instances of all density classes. For a well-known set of benchmark instances, BDDs provide better mean bounds in less time for all but the sparsest class of instances (i.e., all but those with density less than 0.2). We obtained these results using a barrier LP solver that is generally faster than simplex for these instances.

A further advantage of BDD relaxations is that the quality of the bound can be continuously adjusted by controlling the maximum width of the BDD. This allows one to invest as much or little time as one wishes in improving the quality of the bound. In addition, BDD-based bounds can be obtained for combinatorial problems that are not formulated as mixed integer models. Unlike LP relaxations, BDD relaxations do not presuppose that the constraints take the form of linear inequalities.

Table 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

BDD bounds can be rapidly updated during a search procedure, much as the LP can be reoptimized after branching. This is achieved simply by removing arcs of the BDD that correspond to excluded values of the branching variable, and recomputing the shortest (or longest) path. Nonetheless, due to the speed at which BDDs can be constructed, it may be advantageous to rebuild the BDD from scratch, so as to obtain a relaxation that is suited to the current subproblem. One may be able to adjust the BDD width to obtain a bound that is just tight enough to fathom the current node of the search tree, thus saving time. These remain as research issues.

The above results suggest that BDD-based relaxations may have promise as a general technique for bounding the optimal value of discrete problems. The BDD algorithms presented here are relatively simple, compared with the highly developed technology of LP and mixed-integer solvers, and nonetheless improve the state of the art for at least one problem class. Future research may yield improvements in BDD-based bounding and extend its usefulness to a broader range of discrete optimization problems.

References

Akers, S. B. 1978. Binary decision diagrams. *IEEE Transactions on Computers* **C-27** 509–516.

Andersen, H. R., T. Hadzic, J. N. Hooker, P. Tiedemann. 2007. A constraint store based on multivalued decision diagrams. C. Bessière, ed., *Principles and Practice of Constraint Programming (CP 2007)*, *Lecture Notes in Computer Science*, vol. 4741. Springer, 118–132.

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

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

Bergman, D., A. A. Cire, W.-J. van Hoeve, J. N. Hooker. 2012. Variable ordering for the application of BDDs to the maximum independent set problem. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2012)*. Springer, 34–49.

Bergman, D., W.-J. van Hoeve, J. N. Hooker. 2011. Manipulating MDD relaxations for combinatorial optimization. T. Achterberg, J.C. Beck, eds., *CPAIOR*, *Lecture Notes in Computer Science*, vol. 6697. Springer, 20–35.

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

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

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

Grötschel, Martin, László Lovász, Alexander Schrijver. 1993. *Geometric Algorithms and Combinatorial Optimization*, vol. 2. Springer.

Hadzic, T., J. N. Hooker. 2006. Postoptimality analysis for integer programming using binary decision diagrams, presented at GICOLAG workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna. Tech. rep., Carnegie Mellon University.

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

Hadzic, T., J. N. Hooker, B. O'Sullivan, P. Tiedemann. 2008. Approximate compilation of constraints into multivalued decision diagrams. P. J. Stuckey, ed., *Principles and Practice of Constraint Programming (CP 2008)*, *Lecture Notes in Computer Science*, vol. 5202. Springer, 448–462.

Hoda, S., W.-J. van Hoeve, J. N. Hooker. 2010. A systematic approach to MDD-based constraint programming. *Proceedings of the 16th International Conference on Principles and Practices of Constraint Programming*, *Lecture Notes in Computer Science*, vol. 6308. Springer, 266–280.

Hooker, J. N. to appear. Decision diagrams and dynamic programming. C. Gomes, M. Sellmann, eds., *CPAIOR 2013 Proceedings*. Springer.

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

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

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

Loekito, E., J. Bailey, J. Pei. 2010. A binary decision diagram based approach for mining frequent subsequences. *Knowl. Inf. Syst* **24** 235–268.

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

Rebennack, S., M. Oswald, D. Theis, H. Seitz, G. Reinelt, P. Pardalos. 2011. A branch and cut solver for the maximum stable set problem. *Journal of Combinatorial Optimization* **21** 434–457.

Rebennack, S., G. Reinelt, P. Pardalos. 2012. A tutorial on branch and cut algorithms for the maximum stable set problem. *International Transactions in Operational Research* **19** 161–199.

Rossi, F., S. Smriglio. 2001. A branch-and-cut algorithm for the maximum cardinality stable set problem. *Operations Research Letters* **28** 63–74.

Tomita, E., T. Kameda. 2007. An efficient branch-and-bound algorithm for finding a maximum clique, with computational experiments. *Journal of Global Optimization* **111** 95–111.

Wegener, I. 2000. *Branching programs and binary decision diagrams: theory and applications*. SIAM monographs on discrete mathematics and applications, Society for Industrial and Applied Mathematics.