Journal of Heuristics manuscript No.
(will be inserted by the editor)

BDD-Based Heuristics for Binary Optimization

**David Bergman · Andre A. Cire ·
Willem-Jan van Hoeve · Tallys Yunes**

Received: date / Accepted: date

Abstract In this paper we introduce a new method for generating heuristic solutions to binary optimization problems. We develop a technique based on binary decision diagrams. We use these structures to provide an under-approximation to the set of feasible solutions. We show that the proposed algorithm delivers comparable solutions to a state-of-the-art general-purpose optimization solver on randomly generated set covering and set packing problems.

Keywords binary decision diagrams · heuristics · set covering · set packing

1 Introduction

Binary optimization problems are ubiquitous across many problem domains. Over the last fifty years there have been significant advances in algorithms dedicated to solving problems in this class. In particular, general-purpose algorithms for binary optimization are commonly branch-and-bound methods that rely on two fundamental components: a relaxation of the problem, such as a linear programming relaxation of an integer programming model, and heuristics. Heuristics are used to provide feasible solutions during the search for an optimal one, which in practice is often more important than providing a proof of optimality.

David Bergman, Andre A. Cire, Willem-Jan van Hoeve
Tepper School of Business, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213, U.S.A.
E-mail: {dbergman,acire,vanhoeve}@andrew.cmu.edu

Tallys Yunes (corresponding author)
School of Business Administration, University of Miami
Coral Gables, FL 33124-8237, U.S.A.
E-mail: tallys@miami.edu, Phone: +1 (305) 284-5107, Fax: +1 (305) 284-2321

Much of the research effort dedicated to developing heuristics for binary optimization has primarily focused on specific combinatorial optimization problems; this includes, e.g., the set covering problem (Caprara et al, 1998) and the maximum clique problem (Grosso et al, 2008; Pullan et al, 2011). In contrast, general-purpose heuristics have received much less attention in the literature. The vast majority of the general techniques are embodied in integer programming technology, such as the *feasibility pump* (Fischetti et al, 2005) and the *pivot, cut, and dive* heuristic (Eckstein and Nediak, 2007). A survey of heuristics for integer programming is presented by Glover and Laguna (1997a,b) and Berthold (2006). Local search methods for general binary problems can also be found in Aarts and Lenstra (1997) and Bertsimas et al (2013).

We introduce a new general-purpose method for obtaining a set of feasible solutions for binary optimization problems. Our method is based on an under-approximation of the feasible solution set using binary decision diagrams (BDDs). BDDs are compact graphical representations of Boolean functions (Akers, 1978; Lee, 1959; Bryant, 1986), originally introduced for applications in circuit design and formal verification (Hu, 1995; Lee, 1959). They have been recently used for a variety of purposes in combinatorial optimization, including post-optimality analysis (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). The techniques presented here can also be readily applied to arbitrary discrete problems using *multi-valued decision diagrams* (MDDs), a generalization of BDDs for discrete-valued functions.

Our method is a counterpart of the concept of *relaxed* MDDs, recently introduced by Andersen et al (2007) as an over-approximation of the feasible set of a discrete constrained problem. The authors used relaxed MDDs for the purpose of replacing the typical domain store relaxation used in constraint programming by a richer data structure. They found that relaxed MDDs drastically reduce the size of the search tree and allow much faster solution of problems with multiple all-different constraints, which are equivalent to graph coloring problems. Analogous methods were applied to other types of constraints in Hadzic et al (2008) and Hoda et al (2010).

Using similar techniques, Bergman et al (2011) proposed the use of *relaxed BDDs* to derive relaxation bounds for binary optimization problem. The authors developed a general top-down construction method for relaxed BDDs and reported good results for structured set covering instances. Relaxed BDDs were also applied in the context of the maximum independent set problem, where the ordering of the variables in the BDD were shown to have a significant bearing on the effectiveness of the relaxation it provides (Bergman et al, 2012).

We use BDDs to provide heuristic solutions, rather than relaxation bounds. Our main contributions include:

1. Introducing a new heuristic for binary optimization problems;
2. Discussing the necessary ingredients for applying the heuristic to specific classes of problems;

3. Providing an initial computational evaluation of the heuristic on the well-studied set covering and set packing problems. We show that, on a set of randomly generated instances, the solutions produced by our algorithm are comparable to those obtained with state-of-the-art integer programming optimization software (CPLEX).

The remainder of the paper is organized as follows. We begin by defining BDDs in Section 2. Section 3 describes how to generate and use BDDs to exactly represent the set of feasible solutions to a problem. Section 4 details how the algorithm in Section 3 can be modified to provide an under-approximation of the feasible set and to deliver a set of solutions to a problem. We discuss the application of the algorithm to two problem classes in Section 5. Section 6 presents computational experiments.

2 Binary Decision Diagrams

Binary optimization problems (BOPs) are specified by a set of binary variables $X = \{x_1, \ldots, x_n\}$, an objective function $f : \{0,1\}^n \to \mathbb{R}$ to be minimized, and a set of m constraints $C = \{C_1, \ldots, C_m\}$, which define relations among the problem variables. A *solution* to a BOP P is an assignment of values 0 or 1 to each of the variables in X. A solution is *feasible* if it satisfies all the constraints in C. The set of feasible solutions of P is denoted by $\mathrm{Sol}(P)$. A solution x^* is *optimal* for P if it is feasible and satisfies $f(x^*) \leq f(\tilde{x})$ for all $\tilde{x} \in \mathrm{Sol}(P)$.

A *binary decision diagram* (BDD) $B = (U, A)$ for a BOP P is a layered directed acyclic multi-graph that encodes a set of solutions of P. The nodes U are partitioned into $n+1$ layers, $L_1, L_2, \ldots, L_{n+1}$, where we let $\ell(u)$ be the layer index of node u. Layers L_1 and L_{n+1} consist of single nodes; the root r and the terminal t, respectively. The *width* of layer j is given by $\omega_j = |L_j|$, and the *width* of B is $\omega(B) = \max_{j \in \{1,2,\ldots,n\}} \omega_j$. The *size* of B, denoted by $|B|$, is the number of nodes in B.

Each arc $a \in A$ is directed from a node in some layer j to a node in the adjacent layer $j+1$, and has an associated *arc-domain* $d_a \in \{0,1\}$. The arc a is called a *1-arc* when $d_a = 1$ and a *0-arc* when $d_a = 0$. For any two arcs a, a' directed out of a node u, $d_a \neq d_{a'}$, so that the maximum out-degree of a node in a BDD is 2, with each arc having a unique arc-domain. Given a node u, we let $a_0(u)$ be the 0-arc directed out of u (if it exists) and $b_0(u)$ be the node in $L_{\ell(u)+1}$ at its opposite end, and similarly for $a_1(u)$ and $b_1(u)$.

A BDD B represents a set of solutions to P in the following way. An arc a directed out of a node u represents the assignment $x_{\ell(u)} = d_a$. Hence, for two nodes u, u' with $\ell(u) < \ell(u')$, a directed path p from u to u' along arcs $a_{\ell(u)}, a_{\ell(u)+1}, \ldots, a_{\ell(u')-1}$ corresponds to the assignment $x_j = d_{a_j}$, $j = \ell(u), \ell(u)+1, \ldots, \ell(u')-1$. In particular, an r–t path $p = (a_1, \ldots, a_n)$ corresponds to a solution x^p, where $x^p_j = d_{a_j}$ for $j = 1, \ldots, n$. The set of solutions represented by a BDD B is denoted by $\mathrm{Sol}(B) = \{x^p \,|\, p \text{ is an } r\text{–}t \text{ path}\}$. An *exact* BDD B for P is any BDD for which $\mathrm{Sol}(B) = \mathrm{Sol}(P)$.

For two nodes $u, u' \in U$ with $\ell(u) < \ell(u')$, let $B_{u,u'}$ be the BDD induced by the nodes that belong to some directed path between u and u'. In particular, $B_{r,t} = B$. A BDD is called *reduced* if $\mathrm{Sol}(B_{u,u'})$ is unique for any two nodes u, u' of B. The reduced BDD B is unique when the variable ordering is fixed, and therefore the most compact representation in terms of size for that ordering (Wegener, 2000).

Finally, for a large class of objective functions, e.g. additively separable functions, optimizing over the solutions represented by a BDD B can be reduced to finding a shortest path in B. For example, given a real cost vector c and a linear objective function $c^T x$, we can associate an *arc-cost* $c(u, v) = c_{\ell(u)} d_{u,v}$ with each arc $a = (u, v)$ in the BDD. This way, a shortest r–t path corresponds to a minimum cost solution in $\mathrm{Sol}(B)$. If B is exact, then this shortest path corresponds to an optimal solution for P.

Example 1 Consider the following BOP P.

$$\begin{aligned} \text{minimize} \quad & -2x_1 - 3x_2 - 5x_3 - x_4 - 3x_5 \\ \text{subject to} \quad & 2x_1 + 2x_2 + 3x_3 + 3x_4 + 2x_5 \le 5 \\ & x_j \in \{0, 1\}, \quad j = 1, \ldots, 5 \end{aligned}$$

Figure 1 shows an exact reduced BDD for P. The 0-arcs are represented by dashed lines, while the 1-arcs are represented by solid lines. There are 13 paths in the BDD, which correspond to the 13 feasible solutions of this BOP. Assigning arc costs of 0 to all of the 0-arcs and the cost coefficient of x_j to the 1-arcs on layer j, $j = 1, \ldots, 5$, the two shortest paths in the BDD correspond to the solutions $(0, 1, 1, 0, 0)$ and $(0, 0, 1, 0, 1)$, both optimal solutions for P.

3 Exact BDDs

An exact reduced BDD $B = (U, A)$ for a BOP P can be interpreted as a compact search tree for P, where infeasible leaf nodes are removed, isomorphic subtrees are superimposed, and the feasible leaf nodes are merged into t. In principle, B can be obtained by first constructing the branching tree for P and reducing it accordingly, which is impractical for our purposes.

We present here an efficient top-down algorithm for constructing an exact BDD B for P. It relies on problem-dependent information for merging BDD nodes and thus reducing its size. If this information satisfies certain conditions, the resulting BDD is reduced. The algorithm is a *top-down* procedure since it proceeds by compiling the layers of B one-by-one, where layer L_{j+1} is constructed only after layers $L_1, \ldots, L_j$ are completed.

We first introduce some additional definitions. Let $x' = (x'_1, \ldots, x'_j)$, $j < n$, be a *partial solution* that assigns a value to variables $x_1, \ldots, x_j$. We define

$$F(x') = \{x'' \in \{0, 1\}^{n-j} \mid x = (x', x'') \text{ is feasible for } P\}$$

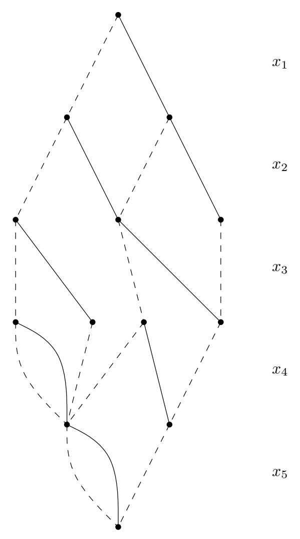


Fig. 1: Reduced BDD for the BOP in Example 1.

as the set of *feasible completions* of x'. We say that two distinct partial solutions x^1, x^2 on variables $x_1, \dots, x_j$ are *equivalent* if $F(x^1) = F(x^2)$.

The algorithm requires a method for establishing when two partial solutions are necessarily equivalent. If this is possible, then the last nodes u, u' of the BDD paths corresponding to these partial solutions can be merged into a single node, since $B_{u,t}$ and $B_{u',t}$ are the same. To this end, with each partial solution x' of dimension k we associate a *state function* $\mathtt{s} : \{0,1\}^k \to S$, where S is a problem-dependent *state space*. The state of x' corresponds to the information necessary to determine if x' is equivalent to any other partial solution on the same set of variables.

Formally, let x^1, x^2 be partial solutions on the same set of variables. We say that the function $\mathtt{s}(x)$ is *sound* if $\mathtt{s}(x^1) = \mathtt{s}(x^2)$ implies that $F(x^1) = F(x^2)$, and we say that $\mathtt{s}$ is *complete* if the converse is also true. The algorithm requires only a sound state function, but if $\mathtt{s}$ is complete, the resulting BDD will be reduced.

For simplicity of exposition, we further assume that it is possible to identify when a partial solution x' cannot be completed to a feasible solution, i.e. $F(x') = \emptyset$. It can be shown that this assumption is not restrictive, but rather makes for an easier exposition of the algorithm. We write $\mathtt{s}(x') = \hat{0}$ to indicate that x' cannot be completed into a feasible solution. If x is a solution to P, we write $\mathtt{s}(x) = \emptyset$ if x is feasible and $\mathtt{s}(x) = \hat{0}$ otherwise.

We now extend the definition of state functions to nodes of the BDD B. Suppose that $\mathtt{s}$ is a complete state function and B is an exact (but not necessarily reduced) BDD. For any node u, the fact that B is exact implies that any two partial solutions $x^1, x^2 \in \mathrm{Sol}(B_{r,u})$ have the same feasible completions, i.e. $F(x^1) = F(x^2)$. Since $\mathtt{s}$ is complete, we must have $\mathtt{s}(x^1) = \mathtt{s}(x^2)$. We hence-

Algorithm 1 Exact BDD Compilation

```
1: Create node $r$ with $\texttt{s}(r) = s_0$
2: $L_1 = \{r\}$
3: for $j = 1$ to $n$ do
4:     $L_{j+1} = \emptyset$
5:     for all $u \in L_j$ do
6:         for all $d \in \{0,1\}$ do
7:             $\texttt{s}_{\text{new}} := \texttt{update}(\texttt{s}(u), d)$
8:             if $\texttt{s}_{\text{new}} \neq \hat{0}$ then
9:                 if $\exists u' \in L_{j+1}$ with $\texttt{s}(u') = \texttt{s}_{\text{new}}$ then
10:                    $b_d(u) = u'$
11:                else
12:                    Create node $u_{\text{new}}$ with $\texttt{s}(u_{\text{new}}) = \texttt{s}_{\text{new}}$
13:                    $b_d(u) = u_{\text{new}}$
14:                    $L_{j+1} \leftarrow L_{j+1} \cup u_{\text{new}}$
```

forth define the state of a node u as $\texttt{s}(u) = \texttt{s}(x)$ for any $x \in \text{Sol}(B_{r,u})$, which is therefore uniquely defined for a complete function $\texttt{s}$.

We also introduce a function $\texttt{update} : S \times \{0,1\} \to S$. Given a partial solution x' on variables $x_1, \ldots, x_j$, $j < n$, and a domain value $d \in \{0,1\}$, the function $\texttt{update}(\texttt{s}(x'), d)$ maps the state of x' to the state of the partial solution obtained when x' is appended with d, $\texttt{s}((x', d))$. This function is similarly extended to nodes: $\texttt{update}(s(u), d)$ represents the state of all partial solutions in $\text{Sol}(B_{r,u})$ extended with value d for a node u.

The top-down compilation procedure is presented in Algorithm 1. We start by setting $L_1 = \{r\}$ and $\texttt{s}(r) = s_0$, where s_0 is an initial state appropriately defined for the problem. Now, having constructed layers $L_1, \ldots, L_j$, we create layer L_{j+1} in the following way. For each node $u \in L_j$ and for $d \in \{0,1\}$, let $\texttt{s}_{\text{new}} = \texttt{update}(\texttt{s}(u), d)$. If $\texttt{s}_{\text{new}} = \hat{0}$ we do not create arc $a_d(u)$. Otherwise, if there exists some $u' \in L_{j+1}$ with $\texttt{s}(u') = \texttt{s}_{\text{new}}$, we set $b_d(u) = u'$; if such a node does not exist, we create node u_{new} with $\texttt{s}(u_{\text{new}}) = \texttt{s}_{\text{new}}$ and set $b_d(u) = u_{\text{new}}$.

Example 2 Consider the following simple binary optimization problem:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \leq 1 \\ & x_j \in \{0,1\}, \quad j = 1,2,3 \end{aligned}$$

We define $\texttt{s}(x)$ to equal the number of variables set to 1 in x. In this way, whenever $\texttt{s}(x^1) = \texttt{s}(x^2)$ for two partial solutions we have $F(x^1) = F(x^2)$. For example, $\texttt{s}((1,0)) = 1$ and $\texttt{s}((0,1)) = 1$, with the only feasible completion being (0).

In addition, we let

$$\texttt{update}(\texttt{s}(u), d) = \begin{cases} \hat{0} & , d = 1 \text{ and } \texttt{s}(u) = 1 \\ 1 & , d = 1 \text{ and } \texttt{s}(u) = 0 \\ s(u) & , d = 0 \end{cases}$$

With this update function, if in a partial solution there is already one variable set to 1, the update operation will assign $\hat{0}$ to the node on the 1-arc to signify that the solution cannot be completed to a feasible solution, and it will assign 1 to the node on the 1-arc to signify that still only one variable is set to 1. On the other hand, if a partial solution has no variable set to 1, the 1-arc will now be directed to a node that has state 1 and the 0-arc will be directed to a node with state 0.

Theorem 1 *Let* $\mathtt{s}$ *be a sound state function for a binary optimization problem* P*. Algorithm 1 generates an exact BDD for* P*.*

Proof We show by induction that at the end of iteration j, the set

$$\bigcup_{u \in L_{j+1}} \mathrm{Sol}(B_{r,u})$$

exactly corresponds to the set of feasible partial solutions of P on $x_1, \dots, x_j$. This implies that after iteration n, $\mathrm{Sol}(B_{r,t}) = \mathrm{Sol}(P)$, since all feasible solutions x have the same state $\mathtt{s}(x) = \emptyset$ and hence L_{n+1} will contain exactly one node at the end of the procedure, which is the terminal t.

Consider the first iteration. We start with the root r and $\mathtt{s}(r) = s_0$, which is the initial state corresponding to not assigning any values to any variables. r is the only node in L_1. When $d = 0$, if there exists no feasible solution with $x_1 = 0$, no new node is created. Hence no solutions are introduced into B. If otherwise there exists at least one solution with $x_1 = 0$, we create a new node, add it to L_2, and introduce a 0-arc from r to the newly created node. This will represent the partial solution $x_1 = 0$. This is similarly done for $d = 1$.

Consider the end of iteration j. Each solution $x' = (x'', d)$ that belongs to $\mathrm{Sol}(B_{r,u})$ for some node $u \in L_{j+1}$ must go through some node $u' \in L_j$ with $b_d(u') = u$. By induction, x'' is a feasible partial solution with $\mathtt{s}(u') = \mathtt{s}(x'') \neq \hat{0}$. But when the arc $a_d(u')$ is considered, we must have $\mathtt{update}(u', d) \neq \hat{0}$, for otherwise this arc would not have been created. Therefore, each solution in $\mathrm{Sol}(B_{r,u})$ is feasible. Since $u \in L_{j+1}$ was chosen arbitrarily, only feasible partial solutions exists in $\mathrm{Sol}(B_{r,u})$ for all nodes $u \in L_{j+1}$.

What remains to be shown is that all feasible partial solutions exist in $\mathrm{Sol}(B_{r,u})$ for some $u \in L_{j+1}$. This is trivially true for the partial solutions $x_1 = 0$ and $x_1 = 1$. Take now any partial feasible solution $x' = (x'', d)$ on the first j variables, $j \geq 2$. Since x' is a partial feasible solution, x'' must also be a partial feasible solution. By induction, x'' belongs to $\mathrm{Sol}(B_{r,u})$, for some $u \in L_j$. When Algorithm 1 examines node u, $\mathtt{update}(\mathtt{s}(u), d)$ must not return $\hat{0}$ because $F(x') \neq \emptyset$. Therefore, the d-arc directed out of u is created, ending at some node $b_d(u) \in L_{j+1}$, as desired. □

Theorem 2 *Let* $\mathtt{s}$ *be a complete state function for a binary optimization program* P*. Algorithm 1 generates an exact reduced BDD for* P*.*

Proof By Theorem 1, B is exact. Moreover, for each j, each node $u \in L_j$ will have a unique state because of line 9. Therefore, any two partial solutions x', x'' ending at unique nodes $u', u'' \in L_j$ will have $F(x') \neq F(x'')$. □

Theorem 3 *Let $B = (U, A)$ be the exact BDD outputted by Algorithm 1 for a BOP P with a sound state function* `s`. *Algorithm 1 runs in time $O(|U|K)$, where K is the time complexity for each call of the* `update` *function.*

Proof Algorithm 1 performs two calls of `update` for every node u added to B. Namely, one call to verify if u has a d-arc for each domain value $d \in \{0, 1\}$. □

Theorem 3 implies that, if `update` can be implemented efficiently, then Algorithm 1 runs in polynomial time in the size of the exact BDD B. Indeed, there are structured problems for which one can define complete state functions with a polynomial time-complexity for `update` (Andersen et al, 2007; Bergman et al, 2011, 2012). This will be further discussed in Section 5.

4 Restricted BDDs

Constructing exact BDDs for general binary programs using Algorithm 1 presents two main difficulties. First, the `update` function may take time exponential in the input of the problem. This can be circumvented by not requiring a complete state function, but rather just a sound state function. The resulting BDD is exact according to Theorem 1, but perhaps not reduced. This poses only a minor difficulty, as there exist algorithms for reducing a BDD B that have a polynomial worst-case complexity in the size of B (Wegener, 2000). A more confining difficulty, however, is that even an exact reduced BDD may be exponentially large in the size of the BOP P. We introduce the concept of *restricted BDDs* as a remedy for this problem. These structures provide an under-approximation, i.e. a subset, of the set of feasible solutions to a problem P. Such BDDs can therefore be used as a generic heuristic procedure for any BOP.

More formally, let P be a BOP. A BDD B is called a restricted BDD for P if $\mathrm{Sol}(B) \subseteq \mathrm{Sol}(P)$. Analogous to exact BDDs, optimizing additively separable objective functions over $\mathrm{Sol}(B)$ reduces to a shortest path computation on B if the arc weights are assigned appropriately. Thus, once a restricted BDD is generated, we can readily extract the best feasible solution from B and provide an upper bound to P.

We will focus on *limited-width* restricted BDDs, in which we limit the size of the BDD B by requiring that $\omega(B) \leq W$ for some pre-set maximum allotted width W.

Example 3 Consider the BOP from Example 1. Figure 2 shows a width-2 restricted BDD. There are eight paths in the BDD, which correspond to eight feasible solutions. Assigning arc costs as in Example 1, a shortest path from the

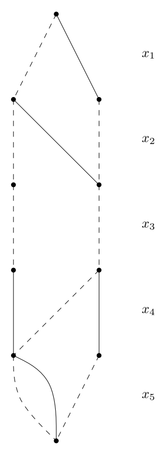


Fig. 2: Width-2 restricted BDD for the BOP presented in Example 1.

Algorithm 2 `delete_nodes`
Insert immediately after line 3 of Algorithm 1.

```
1: if $\omega_j = |L_j| > W$ then
2:     $M :=$ node_select($L_j$)  // where $|M| = \omega_j - W$
3:     $L_j \leftarrow L_j \setminus M$
```

root to the terminal corresponds to the solution $(0, 1, 0, 0, 1)$ with an objective function value of -6. The optimal value is -8.

Limited-width restricted BDDs can be easily generated by performing a simple modification to Algorithm 1. Namely, we insert the procedure described in Algorithm 2 immediately after line 3 of Algorithm 1. This procedure is described as follows. We first verify whether $\omega_j = |L_j| > W$. If so, we delete a set of $|L_j| - W$ nodes in the current layer, which is chosen by a function `node_select`(L_j). We then continue building the BDD as in Algorithm 1.

It is clear that the modified algorithm produces a BDD B satisfying $\omega(B) \leq W$. In addition, it must create a restricted BDD since we are never changing the states of the nodes during the construction, but rather just deleting nodes. Since Algorithm 1 produces an exact BDD, this modified algorithm must produce a restricted BDD.

Theorem 4 describes how the time complexity of Algorithm 1 is affected by the choice of the maximum allotted width W.

Theorem 4 *The modified version of Algorithm 1 for width-W restricted BDDs has a worst-case time complexity of $O(nL + nWK)$, where L and K are the time complexity for each call of the* `node_select` *and* `update` *functions, respectively.*

Proof Because the function `node_select` is called once per layer, it contributes to $O(nL)$ to the overall time complexity. The `update` function is called twice for each BDD node. Since there will be at most $O(nW)$ nodes in a width-W restricted BDD, the theorem follows. $\square$

The selection of nodes in `node_select`(L_j) can have a dramatic impact on the quality of the solutions encoded by the restricted BDD. In fact, as long as we never delete the nodes $u_1, \ldots, u_n$ that are traversed by some optimal solution x^*, we are sure to have the optimal solution in the final BDD.

We observed that the following `node_select` procedure yields restricted BDDs with the best quality solutions in our computational experiments. We are assuming a minimization problem, but a maximization problem can be handled in an analogous way. Each node $u \in L_j$ is first assigned a value $lp(u) = \min f(x) \in \mathrm{Sol}(B_{r,u})$, where f is the objective function of P. This can be easily computed for a number of objective functions by means of a dynamic programming algorithm; for example linear cost functions whose arc weights are as described in Section 2. The `node_select`(L_j) function then deletes the nodes in L_j with the largest $lp(u)$ values. We henceforth use this heuristic for `node_select` in the computational experiments of Section 6. It can be shown that the worst-case complexity of this particular heuristic is $O(W \log W)$.

5 Applications

We now describe the application of restricted BDDs to two fundamental problems in binary optimization: the set covering problem and the set packing problem. For both applications, we describe the problem and provide a sound state function. We then present the `update` operation based on this state function which can be used by the modified version of Algorithm 1.

5.1 The Set Covering Problem

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

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

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

5.1.1 State Function

We now present a sound state function for the purpose of generating restricted BDDs by means of Algorithm 1. Let C_i be the set of indices of the variables that participate in constraint i, $C_i = \{j \mid a_{i,j} = 1\}$, and let $\texttt{last}(C_i) = \max\{j \mid j \in C_i\}$ be the largest index of C_i. We consider the state space $S = 2^{\{1,\dots,m\}} \cup \{\hat{0}\}$. For a partial solution x' on variables $x_1, \dots, x_j$, we write the state function

$$\texttt{s}(x') = \begin{cases} \hat{0}, & \text{if } \exists\, i : \sum_{k=1}^{j} a_{i,k} x'_k = 0 \text{ and } j \geq \texttt{last}(C_i), \\ \left\{ i : \sum_{k=1}^{j} a_{i,k} x'_k = 0 \right\}, & \text{otherwise.} \end{cases}$$

We first argue that the function above assigns a state $\hat{0}$ to a partial solution x' if and only if $F(x') = \emptyset$. Indeed, the condition $\sum_{k=1}^{j} a_{i,k} x'_k = 0$, $j \geq \texttt{last}(C_i)$ for some i implies that all variables that relate to the i-th constraint $\sum_{k=1}^{n} a_{i,j} x_j \geq 1$ are already zero in x', and hence the constraint can never be satisfied. If otherwise that condition does not hold, then $(1, \dots, 1)$ is a feasible completion of x'.

In addition, the following Lemma shows that $\texttt{s}$ is a sound state function for the SCP.

Lemma 1 *Let x^1, x^2 be two partial solutions on variables $x_1, \dots, x_j$. Then, $\texttt{s}(x^1) = \texttt{s}(x^2)$ implies that $F(x^1) = F(x^2)$.*

Proof Let x^1, x^2 be two partial solutions with dimension j for which $\texttt{s}(x^1) = \texttt{s}(x^2) = s'$. If $s' = \hat{0}$ then both have no feasible completions, so it suffices to consider the case when $s' \neq \hat{0}$. Take any completion $\tilde{x} \in F(x^1)$. We show that $\tilde{x} \in F(x^2)$.

Suppose, for the purpose of contradiction, that $(x^2, \tilde{x})$ violates the i^*-th SCP inequality,

$$\sum_{k=1}^{j} a_{i^*,k}\, x_k^2 + \sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k = 0, \tag{1}$$

while

$$\sum_{k=1}^{j} a_{i^*,k}\, x_k^1 + \sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k \geq 1 \tag{2}$$

since $(x^1, \tilde{x})$ is feasible.

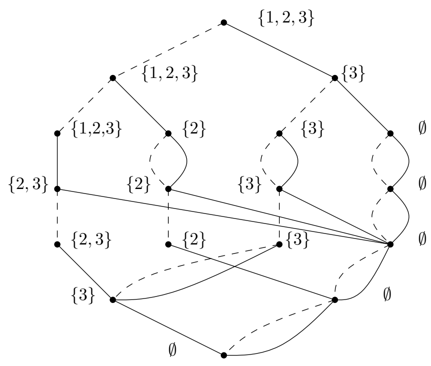
By (1), we have that

$$\sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k = 0 \tag{3}$$

and

$$\sum_{k=1}^{j} a_{i^*,k}\, x_k^2 = 0. \tag{4}$$

The equality (4) implies that $i^* \in \texttt{s}(x^2)$ and therefore $i^* \in \texttt{s}(x^1)$. But then $\sum_{k=1}^{j} a_{i^*,k}\, x_k^1 = 0$. This, together with (3), contradicts (2). $\square$



(a) Exact reduced BDD.

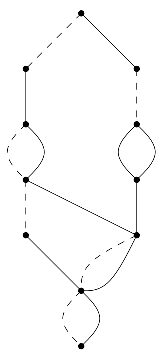

(b) Width-2 restricted BDD.

Fig. 3: Exact and restricted BDDs for the SCP instance in Example 4.

Assuming a partial solution x' on variables $x_1, \ldots, x_j$ and that $s(x') \neq \hat{0}$, the corresponding `update` operation is given by

$$\texttt{update}(\texttt{s}(x'), d) = \begin{cases} \texttt{s}(x') \setminus \{i \mid a_{i,j+1} = 1\}, & d = 1 \\ \texttt{s}(x'), & d = 0, \forall\, i^* \in \texttt{s}(x') : \texttt{last}(C_{i^*}) > j+1 \\ \hat{0}, & d = 0, \exists\, i^* \in \texttt{s}(x') : \texttt{last}(C_{i^*}) = j+1 \end{cases}$$

and has a worst-case time complexity of $O(m)$ for each call.

Example 4 Consider the SCP instance with

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

and

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

Figure 3a shows an exact reduced BDD for this SCP instance where the nodes are labeled with their corresponding states. If outgoing 1-arcs (0-arcs) of nodes in layer j are assigned a cost of c_j (zero), a shortest r–t path corresponds to solution $(1, 1, 0, 0, 0, 0)$ and proves an optimal value of 3. Figure 3b depicts a width-2 restricted BDD where a shortest r–t path corresponds to solution $(0, 1, 0, 1, 0, 0)$, which proves an upper bound of 4.

Example 5 The implication in Lemma 1 is not sufficient as the state function is not complete. Consider the set covering problem

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

and the two partial solutions $x^1 = (1,0)$, $x^2 = (0,1)$. We have $\mathtt{s}(x^1) = \{2\}$ and $\mathtt{s}(x^2) = \{1\}$. However, both have the single feasible completion $\tilde{x} = (1)$.

There are several ways to modify the state function to turn it into a complete one (Bergman et al, 2011). The state function $\mathtt{s}$ can be strengthened to a complete state function. This requires only polynomial time to compute per partial solution, but nonetheless at an additional computational cost. Section 6 reports results for the simpler (sound) state function presented above.

5.2 The Set Packing Problem

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

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \le e \\ & x_j \in \{0,1\}, \quad j = 1, \dots, n \end{aligned}$$

where c, A, and e are as in the SCP. Letting A_j be as in Section 5.1, the SPP asks for the maximum-cost subset $V \subseteq \{1, \dots, n\}$ of the sets A_j such that for all $i, a_{i,j} = 1$ for at most one $j \in V$.

5.2.1 State Function

For the SPP, the state function identifies the set of constraints for which no variables have been assigned a one and could still be violated. More formally, consider the state space $S = 2^{\{1,\dots,m\}} \cup \{\hat{0}\}$. For a partial solution x' on variables $x_1, \dots, x_j$, we write the state function

$$\mathtt{s}(x') = \begin{cases} \hat{0}, & \text{if } \exists\, i : \sum_{k=1}^{j} a_{i,k} x'_k > 1 \\ \left\{ i : \sum_{k=1}^{j} a_{i,k} x'_k = 0 \text{ and } \mathtt{last}(C_i) > j \right\}, & \text{otherwise.} \end{cases}$$

We first argue that the function above assigns a state $\hat{0}$ to a partial solution x' if and only if $F(x') = \emptyset$. Indeed, the condition $\sum_{k=1}^{j} a_{i,k} x'_k > 1$ for some i immediately implies that x' is infeasible; otherwise, $(0, \dots, 0)$ is a feasible completion for x'.

As the following lemma shows, if the states of two partial solutions on the same set of variables are the same, then the set of feasible completions for these partial solutions are the same, thus proving that this state function is sound.

Lemma 2 *Let x^1, x^2 be two partial solutions on variables $x_1, \dots, x_j$. Then, $\mathtt{s}(x^1) = \mathtt{s}(x^2)$ implies that $F(x^1) = F(x^2)$.*

Proof Let x^1, x^2 be two partial solutions for which $\mathtt{s}(x^1) = \mathtt{s}(x^2) = s'$. If $s' = \hat{0}$ then both have empty sets of feasible completions, so it suffices to consider the case when $s' \neq \emptyset$. Take any partial solution $\tilde{x} \in F(x^1)$. We show that $\tilde{x} \in F(x^2)$.

Suppose, for the purpose of contradiction, that $(x^2, \tilde{x})$ violates the i^*-th SPP inequality,

$$\sum_{k=1}^{j} a_{i^*,k}\, x_k^2 + \sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k > 1, \tag{5}$$

while

$$\sum_{k=1}^{j} a_{i^*,k}\, x_k^1 + \sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k \leq 1, \tag{6}$$

since $(x^1, \tilde{x})$ is feasible.

First suppose that $\sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k = 1$. By (6), $\sum_{k=1}^{j} a_{i^*,k}\, x_k^1 = 0$. This implies that $F(x^1)$ contains i^* since no variables in C_{i^*} are set to 1 and there exists $\ell \in C_{i^*}$ with $\ell > j$. Therefore $F(x^2)$ also contains i^*, implying that no variable in C_{i^*} is set to one in the partial solution x^2. Hence $\sum_{k=1}^{j} a_{i^*,k} x_k^2 = 0$, contradicting (5).

Now suppose that $\sum_{k=j+1}^{n} a_{i^*,k}\, \tilde{x}_k = 0$. Then $\sum_{k=1}^{j} a_{i^*,k}\, x_k^2 > 1$, contradicting the assumption that $s' = \mathtt{s}(x^2) \neq \emptyset$. $\square$

Given a partial solution x' on variables $x_1, \ldots, x_j$ with $\mathtt{s}(x') \neq \hat{0}$, the corresponding `update` operation is

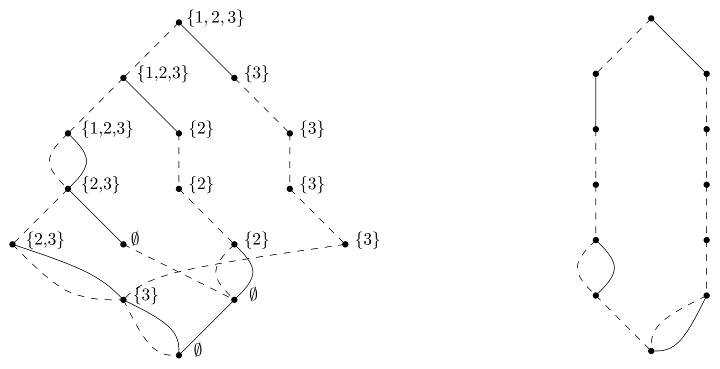
$$\mathtt{update}(s(x'), d) = \begin{cases} \mathtt{s}(x') \setminus \{i \mid \mathtt{last}(C_i) = j+1\}, & d = 0 \\ \mathtt{s}(x') \setminus \{i \mid j+1 \in C_i\}, & d = 1, A_{j+1} \subseteq \mathtt{s}(x') \\ \hat{0}, & d = 1, A_{j+1} \not\subseteq \mathtt{s}(x') \end{cases}$$

and has a worst-case time complexity of $O(m)$ for each call.

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

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

Figure 4a shows an exact reduced BDD for this SPP instance. The nodes are labeled with their corresponding states, and we assign arc costs 1/0 to each 1/0-arc. A longest r–t path, which can be computed by a shortest path on arc weights $c' = -c$ because the BDD is acyclic, corresponds to solution $(0, 0, 1, 0, 1, 1)$ and proves an optimal value of 3. Figure 4b depicts a width-2 restricted BDD where a longest r–t path, for example, corresponds to solution $(1, 0, 0, 0, 0, 1)$, which has length 2.



(a) Exact reduced BDD. (b) Width-2 restricted BDD.

Fig. 4: Exact and restricted BDDs for the SPP instance in Example 6.

Example 7 As in the case of the SCP, the above state function is not complete. For example, consider the problem

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

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

There are several ways to modify the state function above to turn it into a complete one. For example, one can reduce the SPP to an independent set problem and apply the state function defined in Bergman et al (2012). We only consider the sound state function in this work.

6 Computational Experiments

In this section, we perform a computational study on randomly generated set covering and set packing instances. We evaluate our method by comparing the bounds provided by a restricted BDD with the ones obtained via state-of-the-art integer programming technology (IP). We acknowledge that a procedure solely geared toward constructing heuristic solutions for BOPs is in principle favored against general-purpose IP solvers. Nonetheless, we sustain that this is still a meaningful comparison, as modern IP solvers are the best-known general bounding technique for 0-1 problems due to their advanced features and overall performance. This method of testing new heuristics for binary optimization problems was employed by the authors in Bertsimas et al (2013) and we provide a similar study here to evaluate the effectiveness of our algorithm.

The tests ran on an Intel Xeon E5345 with 8 GB of RAM. The BDD code was implemented in C++. We used Ilog CPLEX 12.4 as our IP solver. In particular, we took the bound obtained from the root node relaxation. We set the solver parameters to balance the quality of the bound value and the CPU time to process the root node. The CPLEX parameters that are distinct from the default settings are presented in Table 1. We note that all cuts were disabled, since we observed that the root node would be processed orders of magnitude faster without adding cuts, which did not have a significant effect on the quality of the heuristic solution obtained for the instances tested.

Table 1: CPLEX Parameters

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

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

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

The bandwidth represents the largest distance, in the variable ordering given by the constraint matrix, between any two variables that share a constraint. The smaller the bandwidth, the more structured the problem, in that the variables participating in common constraints are close to each other in the ordering. The *minimum bandwidth problem* seeks to find a variable ordering that minimizes the bandwidth (Martí et al (2008); Corso and Manzini (1999); Feige (2000); Gurari and Sudborough (1984); Martí et al (2001); Piñana et al (2004); Saxe (1980)). This underlying structure, when present in A, can be captured by BDDs, resulting in good computational performance.

6.1 Problem Generation

Our random matrices are generated according to three parameters: the number of variables n, the number of ones per row k, and the bandwidth b_w. For a fixed n, k, and b_w, a random matrix A is constructed as follows. We first initialize A as a zero matrix. For each row i, we assign the ones by selecting k columns uniformly at random from the index set corresponding to the variables $\{x_i, x_{i+1}, \ldots, x_{i+b_w}\}$. As an example, a constraint matrix with $n = 9$, $k = 3$, and $b_w = 4$ may look like

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

Consider the case when $b_w = k$. The matrix A has the *consecutive ones property* and is totally unimodular (Fulkerson and Gross, 1965) and IP finds the optimal solution for the set packing and set covering instances at the root node. Similarly, we argue that an $(m+1)$-width restricted BDD is an exact BDD for both classes of problems, hence also yielding an optimal solution for when this structure is present. Indeed, we show that A containing the consecutive ones property implies that the state of a BDD node u is always of the form $\{j, j+1, \dots, m\}$ for some $j \geq \ell(u)$ during top-down compilation.

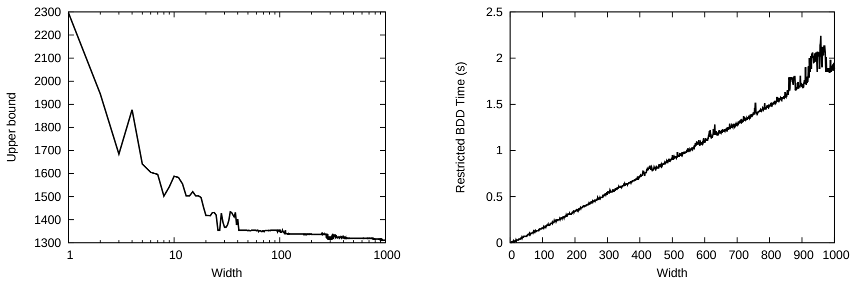
To see this, consider the set covering problem. We claim that for any partial solution x' that can be completed to a feasible solution, $\mathtt{s}(x') = \{i(x'), i(x')+1, \dots, m\}$ for some index $i(x')$, or $\mathtt{s}(x') = \emptyset$ if x' satisfies all of the constraints when completed with 0's.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

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

We note here that generating instances of this sort is not restrictive. Once the bandwidth is large, the underlying structure dissolves and each element of the matrix becomes randomly generated. In addition, as mentioned above, algorithms to solve the minimum bandwidth problem exactly or approximately have been investigated. To any SCP or SPP one can therefore apply these methods to reorder the matrix and then apply the BDD-based algorithm.

6.2 Relation between Solution Quality and Maximum BDD Width

We first analyze the impact of the maximum width W on the solution quality provided by a restricted BDD. To this end, we report the generated bound versus maximum width W obtained for a set covering instance with $n = 1000$, $k = 100$, $b_w = 140$, and a cost vector c where each c_j was chosen uniformly at random from the set $\{1, \dots, nc_j\}$, where nc_j is the number of constraints in



(a) Upper bound. (b) Time.

Fig. 5: Restricted BDD performance versus the maximum allotted width for a set covering instance with $n = 1000$, $k = 100$, $b_w = 140$, and random cost vector.

which variable j participates. We observe that the reported results are common among all instances tested.

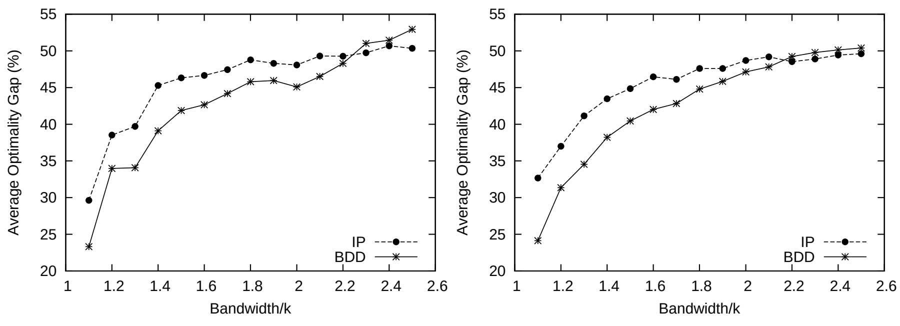
Figure 5a depicts the resulting bounds, where the width axis is in log-scale, and Figure 5b presents the total time to generate the W-restricted BDD and extract its best solution. We tested all W in the set $\{1, 2, 3, \ldots, 1000\}$. We see that as the width increases, the bound approaches the optimal value, with a super-exponential-like convergence in W. The time to generate the BDD grows linearly in W, as expected from the complexity result in Section 4.

6.3 Set Covering

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

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

The feasible solutions are compared with respect to their *optimality gap*. The optimality gap of a feasible solution is obtained by first taking the absolute



(a) Combinatorial. (b) Weighted.

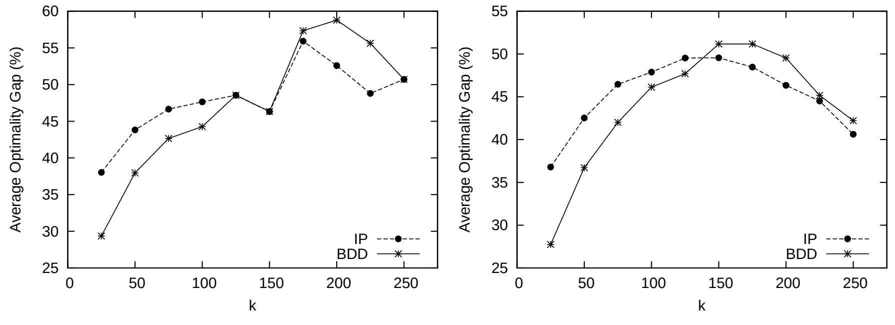
Fig. 6: Average optimality gaps for combinatorial and weighted set covering instances with $n = 500$, $k = 75$, and varying bandwidth.

difference between its objective value and a lower bound to the problem, and then dividing this by the solution's objective value. In both BDD and IP cases, we used the dual value obtained at the root node of CPLEX as the lower bound for a particular problem instance.

The results for the first instance class are presented in Figure 6. Each data point in the figure represents the average optimality gap, over the instances with that configuration. We observe that the restricted BDD yields a significantly better solution for small bandwidths in the combinatorial set covering version. As the bandwidth increases, the staircase structure is lost and the BDD gap becomes progressively worse in comparison to the IP gap. This is a result of the increasing width of the exact reduced BDD for instances with larger bandwidth matrices. Thus, more information is lost when we restrict the BDD size. The same behavior is observed for the weighted set covering problem, although we notice that the gap provided by the restricted BDD is generally better in comparison to the IP gap even for larger bandwidths. Finally, we note that the restricted BDD time is also comparable to the IP time, which is on average less than 1 second for this configuration. This time takes into account both BDD construction and extraction of the best solution it encodes by means of a shortest path algorithm.

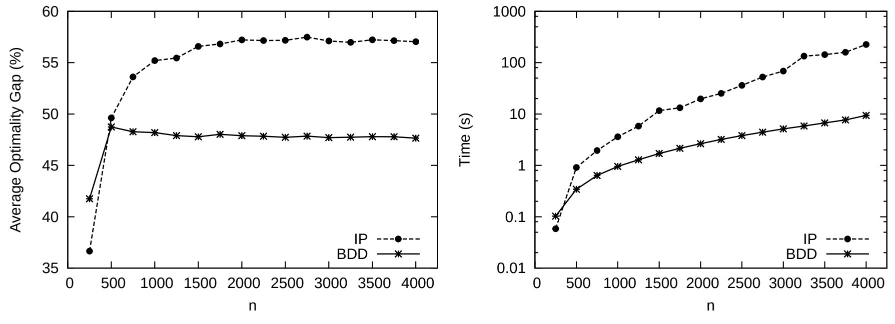
The results for the second instance class are presented in Figure 7. We note that restricted BDDs provide better solutions when k is smaller. One possible explanation for this behavior is that a sparser matrix causes variables to participate in fewer constraints thereby decrease the possible number of BDD node states. Again, less information is lost by restricting the BDD width. Moreover, we note once again that the BDD performance, when compared with CPLEX, is better for the weighted instances tested. Finally, we observe that the restricted BDD time is similar to the IP time, always below one second for instances with 500 variables.

Next, we compare solution quality and time as the number of variables n increases. To this end, we generated random instances with $n \in \{250, 500, 750, \ldots, 4000\}$,



(a) Combinatorial. (b) Weighted.

Fig. 7: Average optimality gaps for combinatorial and weighted set covering instances with $n = 500$, varying k, and $b_w = 1.6k$.

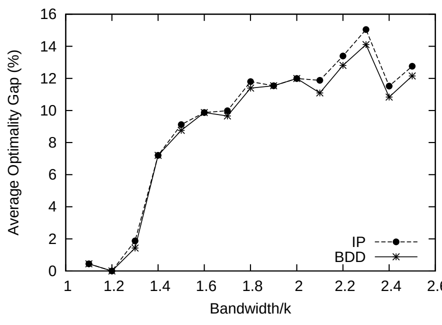


(a) Average Optimality Gap (in %). (b) Time (in seconds).

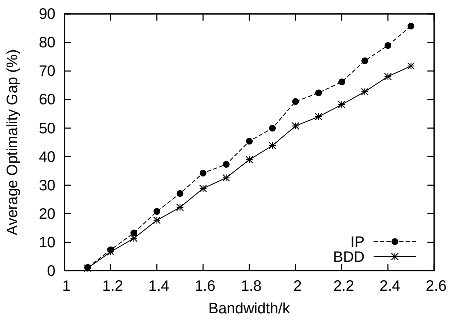
Fig. 8: Average optimality gaps and times for weighted set covering instances with varying n, $k = 75$, and $b_w = 2.2k = 165$. The y axis in the time plot is in logarithm scale.

$k = 75$, and $b_w = 2.2k = 165$. The choice of k and b_w was motivated by Figure 6b, corresponding to the configuration where IP outperforms BDD with respect to solution quality when $n = 500$. As before, we generated 30 instances for each n. Moreover, only weighted set covering instances are considered in this case.

The average optimality gap and time are presented in Figures 8a and 8b, respectively. The y axis in Figure 8b is in logarithm scale. For $n > 500$, we observe that the restricted BDDs yield better-quality solutions than the IP method, and as n increases this gap remains constants. However, IP times grow in a much faster rate than restricted BDD times. In particular, with $n = 4000$, the BDD times are approximately two orders-of-magnitude faster than the corresponding IP times.



(a) Combinatorial.



(b) Weighted.

Fig. 9: Average optimality gaps for combinatorial and weighted set packing instances with $n = 500$, $k = 75$, and varying bandwidth.

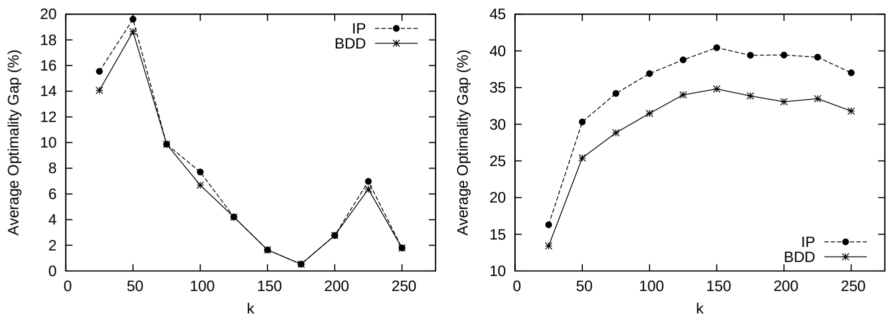
6.4 Set Packing

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

The quality is also compared with respect to the optimality gap of the feasible solutions, which is obtained by dividing the absolute difference between the solution's objective value and an upper bound to the problem by the solution's objective value. We use the the dual value at CPLEX's root node as the upper bound for each instance.

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

The results for the first class of instances are presented in Figure 9. For all tested instances, the solution obtained from the BDD restriction was at least as good as the IP solution for all cost functions. As the bandwidth increases, the gap also increases for both techniques, as the upper bound obtained from CPLEX's root node deteriorates for larger bandwidths. However, the BDD



(a) Combinatorial. (b) Weighted.

Fig. 10: Average optimality gaps for combinatorial and weighted set packing instances with $n = 500$, varying k, and $b_w = 1.6k$.

gap does not increase as much as the IP gap, which is especially noticeable for the weighted case. We note that the difference in times between the BDD and IP restrictions are negligible and lie below one second.

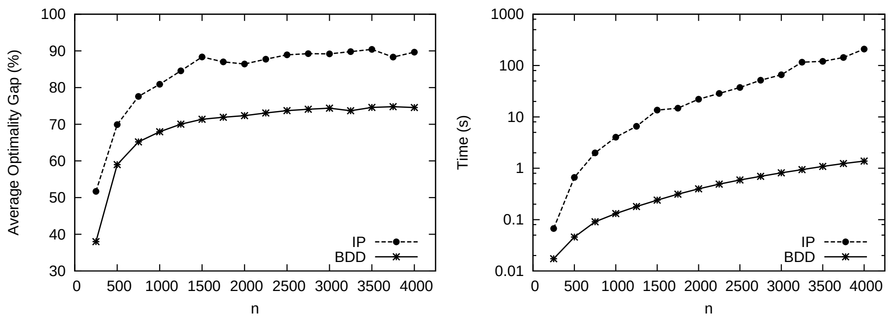
The results for the second class of instances are presented in Figure 10. For all instances tested, the BDD bound was at least as good as the bound obtained with IP, though the solution quality from restricted BDDs was particularly superior for the weighted case. Intuitively, since A is sparser, fewer BDD node states are possible in each layer, implying that less information is lost by restricting the BDD width. Finally, we observe that times were also comparable for both IP and BDD cases, all below one second.

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

The average optimality gap and solving times are presented in Figures 11a and 11b, respectively. Similar to the set covering case, we observe that the BDD restrictions outperform the IP heuristics with respect to both gap and time for this particular configuration. The difference in gaps between restricted BDDs and IP remains approximately the same as n increases, while the time to generate restricted BDDs is orders–of-magnitude less than the IP times for the largest values of n tested.

7 Conclusion

Unlike problem-specific heuristics, general-purpose heuristics for binary optimization problems have received much less attention in the literature. Often, the latter end up incorporated into integer programming software, many of which have dozens of such heuristics at their disposal. With each heuristic



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

Fig. 11: Average optimality gaps and times for weighted set packing instances with varying n, $k = 75$, and $b_w = 2.2k = 165$. The y axis in the time plot is in logarithm scale.

likely to be better suited for BOPs with different mathematical structures, IP solvers typically run many of them at the root node, as well as during search, hoping to find strong primal bounds to help with node pruning and variable fixing. Therefore, it is important for these heuristics to produce high-quality solutions quickly.

We introduce a new structure, restricted BDDs, and describe how they can be used to develop a new class of general-purpose heuristics for BOPs. A restricted BDD is a limited-size directed acyclic multigraph that represents an under-approximation of the feasible set. One of the advantages of representing BOPs with BDDs is that finding the best feasible solution for any separable objective function only requires solving a shortest path problem. Secondly, adapting a generic restricted BDD to a particular problem type is simple; it amounts to defining two criteria used while building the BDD: how to delete nodes from layers that grow beyond the maximum allowed width, and how to combine equivalent nodes in a given layer. Our empirical observations indicate that a good rule of thumb for the first criterion is to keep nodes whose paths to the root of the BDD are the shortest when dealing with minimization objectives, or the longest when dealing with maximization objectives. The second criterion is more problem-specific, as detailed in Section 5, but still often easy to implement.

To test its effectiveness, we apply our restricted-BDD approach to randomly generated set covering and set packing instances, and compare its performance against the heuristic solution-finding capabilities of the state-of-the-art IP solver CPLEX. Our first empirical observation is that, among all instances tested, the quality of the solution obtained by the restricted BDD approaches the optimal value with a super-exponential-like convergence in the value of the maximum BDD width W, whereas the time to build the BDD and calculate the solution only grows linearly in W. For both the set covering and set packing problems we consider combinatorial instances, which have all costs equal to 1, as well as weighted instances, which have arbitrary costs.

For the set covering problem, solutions obtained by the restricted BDD can be up to 30% better on average than solutions obtained by CPLEX. This advantage progressively decreases as either the bandwidth of the coefficient matrix A increases, or the sparsity of A decreases. In general, the BDD performs better on weighted instances. In terms of execution time, the BDD approach has a slight advantage over the IP approach on average, and can sometimes be up to twice as fast.

For the set packing problem, the BDD approach exhibits even better performance on both the combinatorial and weighted instances. Its solutions can be up to 70% better on average than the solutions obtained by CPLEX, with the BDD performing better on weighted instances than on combinatorial instances once again. Unlike what happened in the set covering case, on average, the BDD solutions were always at least as good as the ones produced by CPLEX. In addition, the BDD's performance appears to improve as the bandwidth of A increases. As the sparsity of A changes, the BDD's performance is good for sparse instances, drops at first as sparsity starts to increase, and tends to slowly increase again thereafter. In terms of execution time, the BDD approach can be up to an order of magnitude faster than CPLEX.

In summary, our results indicate that restricted BDDs can become a useful addition to the existing library of heuristics for binary optimization problems. Several aspects of our algorithm may still need to be further investigated, including the application to broader classes of problems and how BDDs can be incorporated into existing complete or heuristic methods. For example, they could be used as an additional primal heuristic during a branch-and-bound search. Moreover, restricted BDDs could also be applied to problems for which no strong linear programming relaxation is known, since they can accommodate constraints of arbitrary form.

References

Aarts E, Lenstra JK (1997) Local Search in Combinatorial Optimization. John Wiley & Sons, New York

Akers SB (1978) Binary decision diagrams. IEEE Transactions on Computers C-27:509–516

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

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

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

Bergman D, van Hoeve WJ, Hooker JN (2011) Manipulating MDD relaxations for combinatorial optimization. In: Achterberg T, Beck J (eds) CPAIOR, Springer, Lecture Notes in Computer Science, vol 6697, pp 20–35

Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2012) Variable ordering for the application of BDDs to the maximum independent set problem. In: Beldiceanu N, Jussien N, Pinson E (eds) 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'12), Springer Verlag, Nantes, France, Lectures Notes in Computer Science, vol 7298, pp 34–49

Berthold T (2006) Primal heuristics for mixed integer programs. Master's thesis, Zuze Institute Berlin

Bertsimas D, Iancu DA, Katz D (2013) A new local search algorithm for binary optimization. INFORMS Journal on Computing 25(2):208–221

Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35:677–691

Caprara A, Fischetti M, Toth P (1998) Algorithms for the set covering problem. Annals of Operations Research 98:2000

Corso GMD, Manzini G (1999) Finding exact solutions to the bandwidth minimization problem. Computing 62(3):189–203

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

Feige U (2000) Approximating the bandwidth via volume respecting embeddings. J Comput Syst Sci 60(3):510–539

Fischetti M, Glover F, Lodi A (2005) The feasibility pump. Math Program 104(1):91–104

Fulkerson DR, Gross OA (1965) Incidence matrices and interval graphs. Pac J Math 15:835–855

Glover F, Laguna M (1997a) General purpose heuristics for integer programming – Part I. Journal of Heuristics 2(4):343–358

Glover F, Laguna M (1997b) General purpose heuristics for integer programming – Part II. Journal of Heuristics 3(2):161–179

Grosso A, Locatelli M, Pullan W (2008) Simple ingredients leading to very efficient heuristics for the maximum clique problem. Journal of Heuristics 14(6):587–612

Gurari EM, Sudborough IH (1984) Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. Journal of Algorithms 5:531–546

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

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

Hoda S, Hoeve WJv, Hooker JN (2010) A systematic approach to MDD-based constraint programming. In: Proceedings of the 16th International Conference on Principles and Practices of Constraint Programming, Springer, Lecture Notes in Computer Science, vol 6308, pp 266–280

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

Lee CY (1959) Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal 38:985–999

Martí R, Laguna M, Glover F, Campos V (2001) Reducing the bandwidth of a sparse matrix with tabu search. European Journal of Operational Research 135(2):450–459

Martí R, Campos V, Piñana E (2008) A branch and bound algorithm for the matrix bandwidth minimization. European Journal of Operational Research 186(2):513–528

Piñana E, Plana I, Campos V, Martí R (2004) GRASP and path relinking for the matrix bandwidth minimization. European Journal of Operational Research 153(1):200–210

Pullan W, Mascia F, Brunato M (2011) Cooperating local search for the maximum clique problem. Journal of Heuristics 17(2):181–199

Saxe J (1980) Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. SIAM J Algebraic Discrete Meth 1:363–369

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